Cours d'introduction à la programmation (en C++)

Structures (en C++)

Jean-Cédric Chappelier
Jamila Sam
Vincent Lepetit

Faculté I&C

Données structurées

Un programme peut avoir à représenter des **données structurées**, par exemple :

Âge
20
35
26
38
22

Nom	Taille	Âge	Sexe
Dupond	1.75	41	M
Dupont	1.75	42	M
Durand	1.85	26	F
Dugenou	1.70	38	M
Pahut	1.63	22	F

Les tableaux permettent de représenter des structures de données *homogènes*, c'est-à-dire des listes constituées d'éléments qui sont tous du *même type*.
Exemples :

```
vector<int> ages;
```

```
array<int, 5> ages;
```

QUID des données *hétérogènes* (c'est-à-dire non homogènes) ?

☞ On les regroupe dans un type composé : les **structures**

A quoi servent les structures ?

On utilise les **structures** lorsque l'on souhaite regrouper des types (pas nécessairement identiques) dans une même entité.

Ce sont une version primitive de la notion « d'*objet* », fondement de la « programmation orientée objet ».

Elles servent à :

- représenter des entités qui doivent être décrites avec plusieurs données
  Par exemple :
  - une personne peut être définie avec son prénom, son nom, sa date de naissance, ...
  - une date est définie par un jour, un mois, et une année

  Elles facilitent la manipulation de telles entités en regroupant les données.
- faire retourner plusieurs valeurs à une fonction
- simplifier la conception et l'écriture des programmes (regroupements conceptuels)

Quelques exemples de structures

```
struct Date
{
  int jour;
  int mois;
  int annee;
};
```

```
struct Etudiant
{
  string nom;
  string section;
  vector<Cours> inscriptions;
  double moyenne;
};
```

```
struct Particule
{
  array<double, 3> position;
  array<double, 3> vitesse;
  double masse;
  double charge;
};
```

Déclaration d'une structure

Pour déclarer un nouveau **type** « structure », on utilise la syntaxe suivante :

```
struct Nom_du_type {
    type_1 identificateur_1 ;
    type_2 identificateur_2 ;
    ...
} ;
```

où
Nom_du_type est le nom que vous souhaitez donner à votre type structuré,
et les
type_i identificateur_i sont les *déclarations des types* et *identificateurs* des **champs** de la structure.

Déclaration d'une structure

Exemples :

```
struct Personne {
  string nom;
  double taille;
  int    age;
  char   sexe; // 'M' ou 'F'
};
```

déclare un nouveau **type**, `Personne`, comme une `struct`ure composée de quatre *champs* : un de type `string`, un autre de type `double`, un troisième de type `int` et un dernier de type `char`.

Autre exemple :

```
struct Complexe {
  double x;
  double y;
};
```

Déclaration d'une structure (2)

Note : Les types des champs d'une structure peuvent aussi être des *types composés*, par exemple des tableaux ou des structures.

Exemple :

```
struct Simple {
  int    souschamp1;
  double souschamp2;
};

struct Compliquee {
  vector<double> champ1;
  int            champ2;
  Simple         champ3;
};
```

Déclaration d'une variable

Une fois le type de la structure déclaré, on peut utiliser son nom comme tout autre type pour déclarer des variables :

Nom_du_type nom_de_la_variable;

Exemples :

```
struct Personne {
  string nom;
  double taille;
  int    age;
  char   sexe;
};

Personne untel;
```

```
struct Complexe {
  double x;
  double y;
};

Complexe z;
```

Initialisation

Les variables de type structure peuvent être initialisées avec la syntaxe suivante :

Type identificateur = { val_1, val_2, ...};

où chaque *val_i* est de type *type_i*, correspondant au champ de même position.

Exemple :

```
struct Personne {
  string nom;
  double taille;
  int    age;
  char   sexe;
};

Personne untel = { "Dupond", 1.75, 41, 'M' };
```

C++11 On peut également utiliser cette syntaxe pour l'affectation :

```
untel = { "Dupond", 1.75, 41, 'M' };
```

Accès aux champs d'une structure

On peut accéder aux champs d'une structure en utilisant la syntaxe suivante :

structure.champ

Exemples :

```
untel.taille = 1.75;

++(untel.age); // déjà un an de plus !

cout << untel.sexe << endl;
```

Exemple complet (1/3)

```
struct Personne {
  string nom;
  double taille;
  int    age;
  char   sexe;
};
```

```
int main()
{
  Personne untel( naissance() );

  anniversaire(untel); // un an de plus

  affiche(untel);
  cout << endl;

  return 0;
}
```

Exemple complet (2/3)

```
Personne naissance() {
  Personne p;

  cout << "Saisie d'une nouvelle personne" << endl;
  cout << "  Entrez son nom : ";
  cin >> p.nom;
  cout << "  Entrez sa taille (m) : ";
  cin >> p.taille;
  cout << "  Entrez son age : ";
  cin >> p.age;
  do {
    cout << "  Homme [M] ou Femme [F] : ";
    cin >> p.sexe;
  } while ((p.sexe != 'F') and (p.sexe != 'M'));

  return p;
}
```

Exemple complet (3/3)

```
void anniversaire(Personne& p) {
  ++(p.age);
}

void affiche(Personne const& p) {
  cout << p.nom    << ", ";
  switch (p.sexe) {
    case 'M': cout << "homme"; break;
    case 'F': cout << "femme"; break;
    default : cout << "alien"; break;
  }
  cout << ", "
       << p.taille << " m, "
       << p.age    << " an";
  if (p.age > 1) {
    cout << 's';
  }
}
```

```
int main()
{
  Personne untel( naissance() );

  anniversaire(untel); // un an de plus

  affiche(untel);
  cout << endl;

  return 0;
}
```

Affectation de structures

Une variable de type composé `struct` peut être directement affectée par une variable du même type

Exemple :

```
Personne p1 = { "Dupond", 1.75, 41, 'M' };
Personne p2;
p2 = p1;
```

La valeur de chaque champ de `p1` est affectée au champ correspondant de `p2`

☞ L'instruction `p2 = p1` est équivalente à la séquence d'instructions

```
p2.nom = p1.nom;       p2.taille = p1.taille;
p2.age = p1.age;       p2.sexe   = p1.sexe;
```

Remarque

Note : l'affection (=) est la seule opération que l'on peut faire de façon globale sur les `struct`.

On **NE** peut **NI** les comparer (`p1 == p2`),

NI les afficher (`cout << p1`) globalement.

Solutions :
Dans ces cas là, il faut le faire champ par champ

Exemple :

```
cout << p1.nom << ", " << p1.taille << ", "
     << p1.age << ", " << p1.sexe ;
```

MIEUX : faire une *fonction* ! !

Retour à l'exemple du début

Les structures sont particulièrement utiles pour les tableaux hétérogènes :

☞ **tableaux de structures**

Exemple :

```
struct Personne {
  string nom;
  double taille;
  int    age;
  char   sexe;
};

vector<Personne> personnes = {
    { "Dupond",  1.75, 41, 'M' },
    { "Dupont",  1.75, 42, 'M' },
    { "Durand",  1.85, 26, 'M' },
    { "Dugenou", 1.70, 38, 'F' },
    { "Pahut",   1.63, 22, 'F' }
};
```

Nom	Taille	Âge	Sexe
Dupond	1.75	41	M
Dupont	1.75	42	M
Durand	1.85	26	M
Dugenou	1.70	38	F
Pahut	1.63	22	F

Fonction à plusieurs valeurs de retour

On sait que les fonctions ne peuvent retourner qu'une seule valeur.

Comment faire lorsque l'on veut « retourner » *plusieurs* valeurs avec une fonction ?

Par exemple, faire retourner le quotient **et** le reste
à la fonction `division_euclidienne(a, b)` ?

Solutions :

1. renvoyer une **structure** contenant les valeurs à retourner ;
2. **passer** les « variables retour » **par référence** et les affecter à l'intérieur de la fonction
3. renvoyer un **tableau dynamique** (`vector`), si les valeurs à retourner sont de même type (homogène)
4. combiner 1 et 3 : structure avec champs `vector` ou (au choix) `vector` de structures (comme dans l'exemple précédent)

Fonction à plusieurs valeurs de retour

Exemple :

1.
```
struct Resultat {
  int quotient;
  int reste;
};
Resultat division_euclidienne(int dividende, int diviseur);
```

2.
```
void division_euclidienne(int dividende, int diviseur,
                          int& quotien, int& reste);
```

3.
```
array<int, 2> division_euclidienne(int dividende, int diviseur);
```
ou
```
vector<int> division_euclidienne(int dividende, int diviseur);
```

Fonction à plusieurs valeurs de retour

Le deux premières solutions ne peuvent être envisagées que pour un *petit nombre* de valeurs à retourner.

La première solution est vraiment la **plus générale**.

La seconde présente l'inconvénient d'avoir à déclarer des variables au préalable et ne peut pas être « chaînée » (`f(g(h(...)))`).

La dernière solution ne fonctionne que pour des valeurs de même type.

Dans le cas où l'on a *beaucoup* de valeurs de *types différents* à retourner, on peut mélanger les solutions 1 et 3 : faire une structure avec des champs de type `vector` ou alors (en fonction du contexte/des besoins) faire un `vector` de structures (comme le tableau de `Personne`s précédent).