Langage C++ et programmation orientée objet

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://clarolineconnect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Plan

Plan

Déclaration d'une structure (1/2)

La déclaration d'une **structure** (type défini par le programmeur) consiste à lister les déclarations des **champs** de la structure.

L'intérêt des structures réside dans le fait que chaque champ peut avoir un type différent, en particulier un type structuré (tableau, chaîne de caractères, et structure) !

```
struct nom_structure
{
    type_champ1 nom_champ1 ;
    type_champ2 nom_champ2 ;
    ...
    type_champn nom_champn ;

} ; // attention au point-virgule!
```

Déclaration d'une structure (1/2)

La déclaration d'une **structure** (type défini par le programmeur) consiste à lister les déclarations des **champs** de la structure.

L'intérêt des structures réside dans le fait que chaque champ peut avoir un type différent, en particulier un type structuré (tableau, chaîne de caractères, et structure) !

```
struct nom_structure
{
    type_champ1 nom_champ1 ;
    type_champ2 nom_champ2 ;
    ...
    type_champn nom_champn ;

} ; // attention au point-virgule!
```

Déclaration d'une structure (2/2)

OK

```
#include <string>
// déclaration de la structure
// personne :
struct personne
{
   int numero; // le premier champ
   string nom, prenom;
   unsigned int age;
   personne * pere, * mere; // autorisé
};
// déclaration de la structure
// trinome :
struct trinome
{
   int numero_trinome;
   personne p1, p2, p3; // on peut avoir une
                        // structure d'un
                        // autre type
   float notes[3]; // on peut avoir un tableau
};
```

PAS OK

```
#include <string>
struct personne
{
   int numero;
   string nom, prenom;
   unsigned int age;
   personne pere, mere; // interdit car
                        // définition
                        // récursive!
   const char sexe; // éviter l'emploi de champ
                    // constant, car si un
                    // champ constant est
                    // présent, alors un
                    // constructeur est
                    // obligatoire
};
```

Déclaration d'une variable du type correspondant

En C

```
struct personne p, p2;
struct trinome t;
```

En C++

```
personne p, p2;
trinome t;
```

Initialisation : les différents cas

- **Si pas de valeur fournie à l'initialisation** :
  - Si variable de la *classe dynamique* (variables allouées par l'utilisateur) : les champs de la structure ne sont pas initialisés (valeurs aléatoires).
  - Si variable de la *classe automatique* (variables locales usuelles) : les champs de la structure ne sont pas initialisés.
  - Si variable de la *classe statique* (variables globales + variables locales précédées du mot clef `static`) : les champs de la structure sont initialisés à zéros.
- **Si une liste de valeurs fournie à l'initialisation** : les valeurs manquantes dans la liste sont complétées par des 0.
- **Si une structure du même type fournie à l'initialisation** : les valeurs des champs de la variable existante sont recopiées dans les champs de la nouvelle variable.

Initialisation : les différents cas

- **Si pas de valeur fournie à l'initialisation** :
  - Si variable de la *classe dynamique* (variables allouées par l'utilisateur) : les champs de la structure ne sont pas initialisés (valeurs aléatoires).
  - Si variable de la *classe automatique* (variables locales usuelles) : les champs de la structure ne sont pas initialisés.
  - Si variable de la *classe statique* (variables globales + variables locales précédées du mot clef `static`) : les champs de la structure sont initialisés à zéros.
- **Si une liste de valeurs fournie à l'initialisation** : les valeurs manquantes dans la liste sont complétées par des 0.
- **Si une structure du même type fournie à l'initialisation** : les valeurs des champs de la variable existante sont recopiées dans les champs de la nouvelle variable.

Initialisation : les différents cas

- **Si pas de valeur fournie à l'initialisation** :
  - Si variable de la *classe dynamique* (variables allouées par l'utilisateur) : les champs de la structure ne sont pas initialisés (valeurs aléatoires).
  - Si variable de la *classe automatique* (variables locales usuelles) : les champs de la structure ne sont pas initialisés.
  - Si variable de la *classe statique* (variables globales + variables locales précédées du mot clef **static**) : les champs de la structure sont initialisés à zéros.
- **Si une liste de valeurs fournie à l'initialisation** : les valeurs manquantes dans la liste sont complétées par des 0.
- **Si une structure du même type fournie à l'initialisation** : les valeurs des champs de la variable existante sont recopiées dans les champs de la nouvelle variable.

Initialisation : les différents cas

- **Si pas de valeur fournie à l'initialisation** :
  - Si variable de la *classe dynamique* (variables allouées par l'utilisateur) : les champs de la structure ne sont pas initialisés (valeurs aléatoires).
  - Si variable de la *classe automatique* (variables locales usuelles) : les champs de la structure ne sont pas initialisés.
  - Si variable de la *classe statique* (variables globales + variables locales précédées du mot clef **static**) : les champs de la structure sont initialisés à zéros.
- **Si une liste de valeurs fournie à l'initialisation** : les valeurs manquantes dans la liste sont complétées par des 0.
- **Si une structure du même type fournie à l'initialisation** : les valeurs des champs de la variable existante sont recopiées dans les champs de la nouvelle variable.

Initialisation : les différents cas

- **Si pas de valeur fournie à l'initialisation** :
  - Si variable de la *classe dynamique* (variables allouées par l'utilisateur) : les champs de la structure ne sont pas initialisés (valeurs aléatoires).
  - Si variable de la *classe automatique* (variables locales usuelles) : les champs de la structure ne sont pas initialisés.
  - Si variable de la *classe statique* (variables globales + variables locales précédées du mot clef **static**) : les champs de la structure sont initialisés à zéros.
- **Si une liste de valeurs fournie à l'initialisation** : les valeurs manquantes dans la liste sont complétées par des 0.
- **Si une structure du même type fournie à l'initialisation** : les valeurs des champs de la variable existante sont recopiées dans les champs de la nouvelle variable.

Initialisation : exemples

```
// **classe dynamique**
personne * pP = new personne ; // aucune initialisation possible (valeur des champs aléatoire)
...
delete pP;

// **classe automatique**
// liste de valeurs pour tous les champs de la variable de type personne
personne p = { 1, "LASTNAME", "FIRSTNAME", 20, NULL, NULL };

// une variable personne pour initialiser
personne p2 = p;

// liste partielle de valeurs
// les champs manquants sont complétés par des zéros
personne p3 = { 1, "LASTNAME" };
personne p4 = { 1, p.nom };

// affectation
personne p4; // déclaration sans initialisation
p4 = p;      // affectation (affectation des champs 1 à 1)
```

Opérateurs d'accès aux champs d'une struct : . et ->

Variable de type *personne*

```
personne p;
p.numero = 5; // p.numero se comporte comme une
              // variable de type int
p.nom = "LASTNAME" ; // possible car p.nom est
                     // de type string qui a
                     // un opérateur "="
                     // surchargé
std::cout << "nom = " << p.nom << std::endl;
p.age = 21 ;
```

Variable de type *personne**

```
personne p;
personne* pt_p = &p;
pt_p->numero = 5; // pt_p->numero se comporte
                  // comme une variable de
                  // type int
pt_p->nom = "LASTNAME" ;
pt_p->prenom = "FIRSTNAME" ;
pt_p->age = 21 ;
```

Variable de type *trinome*

```
trinome t;
t.p1.nom = "LASTNAME1" ;
t.notes[0] = 10 ;
```

Variable de type *trinome**

```
trinome t;
trinome* pt_t = &t;
pt_t->notes[0] = 10 ;
```

Tableaux de structures

```
struct point2D {
                  float x, y;
                };
int main()
{
    point2D courbe2D[10]; // tableau de 10 points 2D (classe automatique)

    courbe2D[4].x=5.2; // accès au champ x du 5ème point2D du tableau courbe2D

    return 0;
}
```

Mode de passage des arguments

Les trois modes de passage d'argument sont autorisés : **par valeur** (*Type* arg), **par référence** (*Type&* arg) et **par adresse** (*Type** pArg).

```
struct point2D {
                  float x, y;
                };
void maFct_valeur(point2D) ;

void maFct_ref(point2D&) ;

void maFct_adresse(point2D*) ;
int main(){
    point2D q ; maFct_valeur(q) ; maFct_ref(q) ; maFct_adresse(&q) ; ...
}
void maFct_valeur(point2D p){
    ... // les modifications de p seront locales à la fonction maFct_valeur
}
void maFct_ref(point2D& p){
    ... // les modifications de p seront transmises à la fonction appelante
}
void maFct_adresse(point2D* pP){
    ... // les modifications de (*pP) seront transmises à la fonction appelante
}
```

retourner une variable d'un type struct

On peut retourner (instruction **return**) :

- Une copie d'une variable de type struct : **passage par valeur**.
- Une référence sur une variable de type struct (*cette variable ne devra pas être de la classe automatique sinon on aura une référence sur une variable qui n'existe plus*) : **passage par référence**.
- Un pointeur (une adresse) sur le début d'un emplacement mémoire associé à une variable de type struct : **passage par adresse**.

retourner une variable d'un type struct

On peut retourner (instruction **return**) :

- Une copie d'une variable de type struct : **passage par valeur**.
- Une référence sur une variable de type struct (*cette variable ne devra pas être de la classe automatique sinon on aura une référence sur une variable qui n'existe plus*) : **passage par référence**.
- Un pointeur (une adresse) sur le début d'un emplacement mémoire associé à une variable de type struct : **passage par adresse**.

retourner une variable d'un type struct

On peut retourner (instruction **return**) :

- Une copie d'une variable de type struct : **passage par valeur**.
- Une référence sur une variable de type struct (*cette variable ne devra pas être de la classe automatique sinon on aura une référence sur une variable qui n'existe plus*) : **passage par référence**.
- Un pointeur (une adresse) sur le début d'un emplacement mémoire associé à une variable de type struct : **passage par adresse**.

Un type struct est "visible" depuis sa déclaration jusqu'à la fin du fichier

Comment utiliser une structure dans plusieurs fichiers ?

La définir dans un fichier d'en-tête (.h, .hpp) et inclure ce fichier d'en-tête dans tous les fichiers sources qui utilisent cette structure : **programmation modulaire**.

Un type struct est "visible" depuis sa déclaration jusqu'à la fin du fichier

Comment utiliser une structure dans plusieurs fichiers ?

La définir dans un fichier d'en-tête (.h, .hpp) et inclure ce fichier d'en-tête dans tous les fichiers sources qui utilisent cette structure : **programmation modulaire**.

Contenu du fichier maStruct.h

```
#ifndef MASTRUCT_H_ /* pour éviter les erreurs de double définition */
#define MASTRUCT_H_

struct maStruct
{
   float x ;
   int n ;
   ...
} ;

#endif /* MASTRUCT_H_ */
```

Alternative aux include guards : #pragma once