Chapitre 13
Les Structures généralisées et Classes

1-*Les Structures généralisées*

✓**Nous pouvons modifier les champs d'une structure un par un, mais c'est souvent un travail monotone.**

✓**Il est préférable de créer des fonctions spéciales, comme ceci :**

```
void ecrit_np_fiche(fiche &f, char  *nouveau_nom,  char *nouveau_prenom)
        {    strcpy(f.nom, nouveau_nom);
             strcpy(f.prenom,nouveau_prenom);
        }
```

✓**Toutefois, de telles fonctions sont très désordonnées si elles sont dispersées à travers le programme.**

✓**Le C++ ajoute au langage C un moyen bien plus simple de traiter ce problème : les fonctions membres.**

1-1 Fonctions membres

Déclaration

✓**Il est bien plus simple de définir en même temps la structure et des fonctions qui agissent sur elle.**

✓**Pour cela, il suffit de déclarer des fonctions membres, appelées aussi méthodes :**

```
struct fiche {    char *nom, *prenom;
                  int age;
                  void ecrit_np(char *nouv_nom, char *nouv_pre);
                };
```

✓**Le compilateur distingue ce membre d'une donnée usuelle à cause des parenthèses.**

✓**Observons un point important : la structure elle-même n'a pas été passée en paramètre.**

✓**En effet, une fonction membre reçoit toujours l'objet par lequel elle est appelée, sous la forme d'un paramètre implicite de type pointeur, nommé *this*.**

Implantation

✓**L'implantation de la fonction membre se fait comme suit :**

```
void fiche::ecrit_np(char *nouv_nom, char  *nouv_pre)
                    {    strcpy(this->nom,nouv_nom);
                         strcpy(this->prenom,nouv_pre);
                    }
```

✓**Noter que le nom de la méthode est précédé du nom de la structure suivi par l'opérateur de résolution de portée :: . On indique ainsi au compilateur qu'il s'agit de la fonction membre définie dans la structure fiche.**

✓**En effet, d'autres structures pourraient avoir une fonction membre du même nom, et il peut y avoir aussi une fonction normale ayant ce nom; en outre le compilateur sait ainsi immédiatement qu'il doit passer un paramètre implicite fiche *this dans la fonction. C'est pourquoi le nom de la structure est obligatoire : *il ne doit jamais être omis, même s'il n'y a qu'une fonction portant ce nom dans tout le programme.***

✓**Nous voyons ici l'usage du paramètre caché this.**

✓**Cependant, cette écriture est assez lourde. Il est permis de l'abréger ainsi :**

```
void fiche::ecrit_np(char *nouv_nom, char  *nouv_pre)
{    strcpy(nom,nouv_nom);
     strcpy(prenom,nouv_pre);
}
```

✓**En effet, toutes les fonctions membres « connaissent » automatiquement le nom de tous les membres (fonctions et données) de la structure.**

✓**De ce fait, on utilise assez peu le paramètre *this* explicitement, sauf lorsqu'on souhaite connaître l'adresse de la structure (c'est pourquoi *this* est un pointeur, et non une référence).**

Appel d'une fonction membre

✓ **On appelle une fonction membre avec l'opérateur . (point), de la même façon qu'on désigne une donnée membre :**

```
fiche employe;
employe.ecrit_np("AAAA", "BBBB");
```

✓ **ou l'opérateur -> s'il s'agit d'un pointeur :**

```
fiche *pempl;
pempl->ecrit_np("AAAA", "BBBB");
```

✓ **Dans le premier cas, le paramètre *this* passé à la méthode ecrit_np est &employe, dans le second cas c'est pempl.**

Ces deux appels sont équivalents à :

```
ecrit_np_fiche(employe, "AAAA", "BBBB");
ecrit_np_fiche(*pempl, "AAAA", "BBBB");
```

2- Les classes

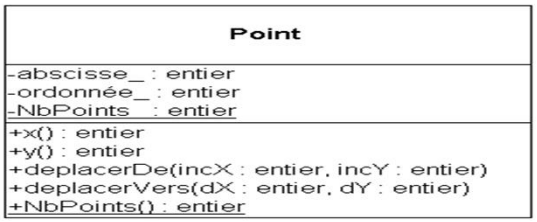
2-1 Représentation des classes

✓**En C++, une classe est représentée sous la forme d'une forme particulière de structure (struct) qui rassemble à la fois les attributs et les méthodes.**

✓**En C++, l'interface d'une classe est constituée de la liste des méthodes déclarées visibles.**

✓**La figure ci-dessous illustre la transposition d'un diagramme de classe en sa représentation C++.**

Description UML



Déclaration de la classe

Constructeur

Méthodes

Membres de classe

Attributs

```
class Point
{
  public:
    Point(int absc, int ordo);
    int x(void) const;
    int y(void) const;
    void deplacerDe(int incX, int incY);
    void deplacerVers(int dX, int dY);

    static int NbPoints(void);
  private:
    int abscisse_;
    int ordonnee_;

    static int NbPoints_;
};
```

Définition de l'attribut de classe

```
int Point::NbPoints_=0;
```

Définition des méthodes

```
Point::Point(int absc, int ordo)
{
  abscisse_=absc;
  ordonnee_=ordo;
}

int Point::x(void) const
{
  return abscisse_;
}

int Point::y(void) const
{
  return ordonnee_;
}

static int Point::NbPoints(void)
{
  return NbPoints_;
}

void Point::deplacerDe(int incX, int incY)
{
  abscisse_+=incX;
  ordonnee_+=incY;
}

void Point::deplacerVers(int dX, int dY)
{
  abscisse_=dX;
  ordonnee_=dY;
}
```

Quelques explications :

- ✓La déclaration de la classe commence par le mot clef class et est encadrée par une paire d'accolades.
- ✓L'accolade finale doit impérativement est suivie d'un point virgule.
- ✓Pour l'instant, il vous suffit de savoir que les membres déclarés après le mot clef public forment l'interface de la classe alors que ceux suivant private sont invisibles de l'utilisateur.
- ✓L'ordre de déclaration des méthodes et des attributs est laissé au libre arbitre du programmeur.
- ✓Toutefois, il est d'usage de déclarer les attributs à la fin car la partie la plus importante d'une classe du point de vue de l'utilisateur est son interface.
- ✓En conséquence, il est important de placer les méthodes au début.

- ✓La déclaration des attributs est semblable à la déclaration d'une variable alors que celle d'une méthode ressemble au prototype d'une fonction ! Néanmoins, il ne faut pas oublier que chaque méthode possède un argument caché : l'objet sur lequel elle est invoquée.
- ✓Lors de l'implémentation des méthodes, il est nécessaire de préfixer le nom de la méthode implémentée du nom de la classe suivi de '::'.
- ✓Par exemple, l'implémentation de la méthode *deplacerVers* de la classe *Point* se fait en spécifiant *Point::deplacerVers*
- ✓ Le constructeur est une méthode particulière qui porte le même nom que la classe et dont le but est d'initialiser les attributs lors de la création d'un objet (plus de détail dans le chapitre suivant).
- ✓ Les méthodes x et y sont déclarées constantes à l'aide du mot clef *const.* Cela signifie que leur code n'affecte en aucune manière la valeur des attributs de l'objet cible.

2-2 Organisation du code source

✓ **A l'exception de classes fortement reliées, il est conseillé de placer une seule classe par fichier source.**

✓ **En fait, une classe a même besoin de 2 fichiers sources :**

- **un fichier header (.h, .H, .hxx, .hh ou .hpp selon les environnements) où l'on place la déclaration de la classe.**
- **un fichier de définition des méthodes et des variables de classe (.C, .cpp, .cc ou .cxx selon les environnements).**

✓**Afin d'éviter les inclusions redondantes, on place des balises de compilation avec des #ifndef comme dans l'exemple suivant:**

```
#ifndef __NOM_DE_LA_CLASSE_H
#define __NOM_DE_LA_CLASSE_H
        // Placer ici les inclusions et les déclarations externes nécessaires
        // Placer ici la déclaration de la classe
Class NomDeLaClasse
                        {

                        };// Ne pas oublier « ; »

          // Sauver sous le nom : nom_de_la_classe.h
#endif
```

[Programme: Structure du fichier de déclaration d'une classe]

Le fichier de définition sera :

```
#include "nom_de_la_classe.h"
        // autres inclusions nécessaires
       // Définitions des variables de classe
      // Définitions des méthodes
```

[Programme: Structure du fichier de définition d'une classe]

2.3 Les modificateurs d'accès

- ✓ ***Les mots clefs public et private sont des modificateurs d'accès. Leur portée s'étend jusqu'au prochain modificateur.***
- ✓ ***Le modificateur par défaut est private. Les membres déclarés private ne sont visibles que par les méthodes de la classe elle même.***
- ✓ ***En revanche, tout membre déclaré public aura une visibilité universelle.***
- ✓ ***Le respect du principe d'encapsulation impose donc que :***
  - ➢ ***Tout attribut d'instance ou de classe sera déclaré private (abstraction de données)***
  - ➢ ***Toute méthode non nécessaire à l'utilisateur sera déclarée private***
  - ➢ ***Toute méthode que vous souhaitez rendre disponible à l'utilisateur, et qui définit donc l'interface de la classe est à déclarer public***

✓**Pour résumer, seules les méthodes de l'interface doivent être déclarées public, tout le reste doit être private.**

✓**Nous verrons qu'un troisième modificateur de visibilité, nommé protected sera utilisé lorsque nous traiterons de l'héritage.**

2.4 Déclaration et définition des membres de classes

✓**Les membres de classe sont déclarés avec le mot clef *static*. Contrairement aux modificateurs d'accès, *static* n'a d'effet que sur une seule ligne de déclaration.**

✓**Particularité du C++ : la déclaration d'une donnée membre *static* ne lui affecte pas d'adresse, il faudra définir cette donnée membre par ailleurs dans le fichier de définitions. C'est précisément le rôle de la définition**

int Point::NbPoints_=0

qui définit la donnée membre de classe NbPoints et lui affecte la valeur initiale 0.

✓**Rappelons que si les méthodes d'instance peuvent très bien utiliser les attributs de classe en plus des attributs d'instance, les méthodes de classe elles ne peuvent qu'utiliser les attributs de classe.**

2.5 La résolution de portée

✓**Vous aurez remarqué un opérateur particulier "::" appelé « opérateur de résolution de portée ». Il sert à désigner à quelle classe appartient une méthode ou un attribut.**

✓**Cet opérateur est nécessaire à la définition des méthodes car le langage C++ ne vous oblige pas à respecter la règle de séparation de l'implémentation dans différents fichiers.**

✓**Aussi, la définition de chaque méthode doit être précédée du nom de la classe à laquelle elle se rattache et de l'opérateur de résolution de portée.**

✓**Par exemple, supposons que dans le même fichier, vous vouliez implémenter la méthode met1 de la classe A ainsi que la méthode met2 de la classe B, alors, vous auriez à spécifier :**

```
typeRetour A::met1(paramètres)
{
 // code d'implémentation
}
typeRetour B::met2(paramètres)
{
 // code d'implémentation
}
```

Programme: Utilisation de l'opérateur de résolution de portée

✓ **L'accès à un membre de classe peut se faire en utilisant ::identificateur**

2.6 Les méthodes inline

✓ ***Le problème des fonctions est que la séquence appel de fonction + retour de fonction, dont la durée et la longueur de code sont négligeables, devient ici une grande partie de l'opération effectuée.***

✓ ***Or, vu la petite taille du code généré, il serait préférable que celui-ci soit placé directement à l'endroit où la fonction est appelée.***

✓ ***Les méthodes inline ont été inventées pour cela. La principale caractéristique des méthodes inline est leur faculté à développer leur code en lieu et place d'un appel à la manière d'une macro.***

✓**Ce qui signifie, un gain du temps nécessaire à l'appel d'une fonction, ainsi que l'accès à un attribut ne coûte plus rien.**

✓**Il y a néanmoins un inconvénient, comme tout appel est remplacé par le développement du code de la méthode, il en résulte un accroissement de la taille du code cible.**

✓**Aussi, ces méthodes doivent être limitées à quelques instructions sous peine d'accroître quasi indéfiniment la taille de l'exécutable.**

✓**En outre, certains compilateurs refusent de mettre en ligne les méthodes qui contiennent des boucles.**

Comment une méthode devient elle inline ?

il existe 2 manières de le faire :

- **Décrire l'implémentation de la méthode au niveau de sa déclaration. C'est la manière la plus simple mais elle présente un défaut : ne pas séparer l'implémentation de la déclaration.**

- **Par opposition aux méthodes décrites dans la déclaration d'une classe, on appelle méthode déportée une méthode dont le code n'est pas transcrit dans la déclaration de sa classe mais en dehors.**

- **Notons qu'il est néanmoins possible de faire développer inline une méthode déportée. Il faut alors préfixer sa déclaration ainsi que sa définition du mot clef inline.**

➢ **En outre, il faut donner son implémentation dans le fichier de déclaration à la suite de la déclaration de la classe.**

➢ **En effet, si vous souhaitez que le compilateur puisse développer le code de la méthode sur le lieu de l'appel, il faut qu'il connaisse sa taille.**

➢ **Par exemple, nous allons réécrire la classe Point en utilisant des méthodes inline.**

✓**Nous allons mettre inline les deux méthodes d'accès aux attributs ainsi que le constructeur.**

✓**Afin d'exploiter toutes les possibilités, les méthodes d'accès aux attributs seront placées inline dans la déclaration alors que le constructeur sera mis inline externe.**

✓**En respectant la structure en 2 fichiers, nous obtenons :**

```
#ifndef __POINT_H
#define __POINT_H
class Point
{
  public:

inline Point(int absc, int ordo); // Constructeur déclaré inline
                                  //le compilateur va rechercher le code plus loin
int x(void) const                 // Déclaration et définition inline
    {
       return abscisse_;
    }
int y(void) const                 // Déclaration et définition inline
     {
       return ordonnee_;
     }
void deplacerDe(int incX, int incY); //Méthode NON inline
void deplacerVers(int dX, int dY);   // méthode NON inline
static int NbPoints(void);

private:
     int abscisse_;
     int ordonnee_;
     static int NbPoints_;
};
// Definition inline deportee du constructeur
inline Point::Point(int absc, int ordo)
{
  abscisse_=absc;
  ordonnee_=ordo;
}
#endif
```

Programme Point.h : fichier d'entête de la classe Point

```cpp
#include "Point.h"

// Definition de l'attribut statique NbPoints_
// notez que l'on ne repete pas le mot clef static
// la valeur d'initialisation est OBLIGATOIRE

int Point::NbPoints_=0;

void Point::deplacerDe(int incX, int incY)
{
   abscisse_+=incX;
   ordonnee_+=incY;
}

void Point::deplacerVers(int dX, int dY)
{
   abscisse_=dX;
   ordonnee_=dY;
}
```

Programme Point.cpp : fichier d'implémentation de la classe Point