partie 2 : Premiers concepts autour de la programmation objet

MOOC Langage C++ INSA

Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7

1 Notion d'objets

1.1 Premières définitions

La notion d'objet dans un programme correspond à un ensemble de (petites) entités interagissant entre elles. Chaque entité possède une certaine autonomie. G. Booch [**?**] définit l'objet comme une entité possédant :

- **une identité** : elle caractérise l'existence propre de l'objet. Elle permet de distinguer tout objet de façon non ambiguë. L'identité est en général implicite dans le langage,
- **un état** : il regroupe les valeurs instantanées de tous les attributs d'un objet. Un attribut est une information qui qualifie l'objet qui le contient,
- **un comportement** : il regroupe les compétences d'un objet. Il décrit les actions et les réactions de cet objet. Chaque atome de comportement est appelé 'opération' ou 'méthode'. Les opérations d'un objet sont déclenchées suite à une stimulation externe représentée sous la forme d'un message envoyé par un autre objet. Ceci correspond à un appel de l'opération avec ou sans arguments.

Exemple:
La 2 CV de monsieur X peut être un objet. Elle possède des attributs : sa date de mise en circulation, son propriétaire, sa couleur, sa vitesse, etc. Ses attributs constituent son état. Ce dernier peut changer ; on peut changer sa couleur. Cette 2 CV peut être démarrée, on peut accélérer, freiner, etc. Ceci va définir les actions extérieures qu'elle peut subir.

Une classe décrit le domaine de définition d'un ensemble d'objets. Chaque objet appartient à une classe. Les objets sont construits par **instanciation** d'une classe. Une classe possède trois parties : son nom, ses attributs et ses opérations.

Exemple:
La 2 CV de monsieur X est différente de la 2 CV de monsieur Y mais elles appartiennent toutes les deux à la famille ou classe des 2 CV.

1.2 Avantage de la programmation objet

La programmation objet permet de rendre moins long (et donc moins coûteux) et plus sûr le développement et la maintenance des logiciels. Ceci tient du fait que les programmes ou bibliothèques

écrits en langage objet sont plus facilement extensibles et réutilisables. On s'appuie sur le concept de modularité. Ce dernier permet :

- **la décomposabilité** favorise la décomposition du problème en sous-problèmes,
- **la composabilité** favorise la production de "briques" facilement combinables,
- **la compréhensibilité** fournit des modules compréhensibles car élémentaires,
- **la continuité** permet de construire des architectures telles qu'un petit changement amène des changements limités à un petit nombre de modules voisins,
- **la protection** assure qu'une erreur dans un module ne doit avoir de répercussions que dans un petit nombre de modules voisins.

2 Cheminement du C++ vers la programmation objet

Nous décrivons dans la suite quelques ajouts au C pour permettre de construire un langage orienté objet comme le C++.

2.1 Structures Actives

Le C++ autorise la déclaration de fonctions encapsulées dans les structures qu'elles manipulent. Il faut bien noter l'appel à la fonction `norme` qui se fait maintenant avec la notation pointée.

Exemple:

```
#include <iostream>
#include <math.h>
using namespace std;
struct complexe {
          float r,i;
          float norme(complexe x) {
                    return sqrt (x.r*x.r + x.i*x.i);
          }
};
int main (){
          complexe x;
          x.r = 1.1;
          x.i = 2.0;
          cout << "norme x = (" << x.r << "," << x.i << ") = "
                    << x.norme(x) << endl;
                    /* endl permet le retour à la ligne*/
          return 0;
}
```

Ce type de déclaration permet à la fonction déclarée dans la structure de pouvoir accéder directement aux champs de la structure. Il n'est donc plus nécessaire de passer en paramètre de la fonction `norme` le complexe sur lequel on travaille. Il n'y a pas d'ambiguïté. Les champs `r` et `i` que l'on manipule sont ceux du complexe qui a appelé la fonction `norme`. Ce qui se traduit par l'écriture suivante :

Exemple:

```
#include <iostream>
#include <math.h>
using namespace std;
struct complexe {
          float r,i;
```

```
          float norme() {
                    return sqrt (r*r + i*i);
          }
};
int main (){
          complexe x;
          x.r = 1.1;
          x.i = 2.0;
          cout << "norme x = (" << x.r << "," << x.i << ") = "
                    << x.norme() << endl;
          return 0;
}
```

2.2 Protection des structures

À l'aide d'un exemple, nous donnons dans la suite les principes fondamentaux de la programmation objet avec la notion de constructeur et de protection.

2.2.1 Problème d'exécution

Voici un exemple montrant une erreur classique de programmation :

Exemple:

```
#include <iostream>
#include <math.h>
using namespace std;
struct complexe {
          float r,i;
          float norme() {
                    return sqrt (r*r + i*i);
          }
};
int main (){
          complexe y; // non initialisé
          cout << y.norme() << endl;
          //l'exécution affiche n'importe quoi
          return 0;
}
```

Le problème provient d'une variable qui est utilisée sans avoir été initialisée (dans l'exemple la variable y). On peut intégrer dans la structure un mécanisme permettant d'éviter ce type d'erreur, par exemple en définissant une sentinelle. Cette dernière correspond à une variable booléenne indiquant si l'objet est initialisé ou non, dans l'exemple suivant, elle est nommée "vide".

Exemple:

```
#include <iostream>
#include <math.h>
using namespace std;
struct complexe {
     float r,i;
     bool vide;
     float norme() {
          if (vide == true) {
               cout << ``norme() appelée sur objet non initialisé";
```

```
                exit(-2); }
          else {
                return sqrt (r*r + i*i);}
     }
     void set (float x,float y) {
          r = x; i = y;
          vide = false;
     }

   };

   int main (){
     { // fonctionnement correct à l'exécution
          complexe y;
          y.set(2,3);
          cout << y.norme() << endl;
          // affiche : 3.60555      }
     { // fonctionnement incorrect à l'exécution
          complexe y1;
          cout << y1.norme() << endl;
     //exécution affiche n'importe quoi

        }
     return 0;
}
```

Pourtant, cela ne marche toujours pas car la variable "vide" de y1 n'est pas initialisée à "true" par défaut. Elle peut prendre n'importe quelle valeur à sa création et par exemple "false". Une initialisation par défaut est impossible pour les membres d'une structure.

2.2.2 Notion de constructeur

La solution du problème précédent passe par la définition explicite d'un constructeur. Ce dernier correspond à une opération particulière de la structure qui sera appelée automatiquement à la déclaration de la variable. Cette opération doit avoir le même nom que la structure.

Exemple:

```
#include <iostream>
#include <math.h>
using namespace std;
struct complexe {
     float r,i;
     bool vide;
     //constructeur
     complexe() {
          vide = true;
     }
     float norme() {
          if (vide == true) {
                cout <<"norme() appelée sur objet non initialisé";
                exit(-2);}
          else {
                return sqrt (r*r + i*i);}
     }
     void set (float x,float y) {
          r = x; i = y;
          vide = false;
     }
};
```

```
   int main (){
     complexe y;
     cout << y.norme() << endl;
     //l'exécution entraîne exit
     return 0;
}
```

Le problème d'initialisation est résolu mais il y a encore un risque de se tromper. En effet, la variable sentinelle "vide" peut être modifiée par le programme principal en utilisant la structure "complexe".

2.2.3 Visibilité des champs

Pour éviter que la variable sentinelle "vide" puisse être modifiée par le programme utilisant la structure "complexe", il faut limiter la visibilité du champ "vide".

Exemple:

```
...
int main(){
     complexe y;
     y.vide = false; // SOURCE D'ERREUR
     cout << y.norme() << endl;
     // l'exécution donne de nouveau n'importe quoi
     return 0;
}
```

Le C++ offre la posssibilité de rendre accessible ou non certaines données et fonctions internes aux structures. Il existe deux niveaux de visibilité des membres d'une structure :

- '`private`' visible uniquement à l'intérieur de la structure
- '`public`' visible partout (par défaut).

Il existe un troisième niveau de protection qui sera présenté plus loin.

Exemple:

```
#include <iostream>
#include <math.h>
using namespace std;
struct complexe {
public: /* utilisable en dehors de la stucture*/
     float r,i;
     //constructeur
     complexe() {vide = true;}
     float norme() {
          if (vide == true) {
               cout <<"norme() appelée sur un objet non initialisé";
               exit(-2);}
          else {
               return sqrt (r*r + i*i);}
     }
     void set (float x,float y) {
          r = x; i = y;
          vide = false;
     }
```

```
private : // protège la variable vide
     bool vide;
};
int main (){
     complexe y;
     y.vide = false; // ***ERREUR DE COMPILATION
     cout << y.norme() << endl;
     return 0; }
```

L'erreur d'exécution n'a plus lieu, désormais la compilation détectera le problème de programmation.

3 Classes

La notion de classe est fonctionnellement équivalente à la notion de structure. Le C++ introduit tout de même le mot clé '`class`'.

3.1 Protection

La différence entre une classe et une structure est que, par défaut, le contenu de la classe est privé alors que celui de la structure est public. Ce choix pour le monde de la programmation objet vient de la volonté d'éviter les oublis pour la protection des données. Si rien n'est modifiée, la protection est maximum. On parlera aussi plutôt d'attributs pour une classe et de champs pour une structure.

Exemple:

Caractéristiques d'une classe :

```
class exemple_class {
// tout ce qui suit est par défaut privé
int i;
int f() {...}
//...jusqu'à
public : // tout ce qui suit est maintenant public
//...jusqu'à
private: //tout ce qui suit est de nouveau privé
//...
};
```

Caractéristiques d'une structure :

```
struct exemple_struct {
// tout ce qui suit est par défaut public
int i;
int f() {...}
//...jusqu'à
private : // tout ce qui suit est maintenant privé
//...jusqu'à
public : // tout ce qui suit est de nouveau public
//...
};
```

Il existe un troisième type de protection : `protected`. Dans ce cas, les attributs et opérations visées sont accessibles aux fonctions membres de la classe ainsi qu'aux fonctions membres des classes dérivées. La notion de classes dérivées sera expliquée plus loin (cf chapitre **??**). On peut cependant dire que cela permet de construire de nouvelles classes en s'appuyant sur des classes existantes.

Remarque:

Ces notions de protection sont différentes de la notion de visibilité. Un nom peut être visible, bien qu'on ne puisse pas y avoir accès.

Le choix des attributs privés se fait selon ces critères :

- Premier critère : une donnée (respectivement une fonction) doit être privée si sa modification directe peut entraîner une inconsistance dans la description de l'objet,
- Deuxième critère : toutes les données qui représentent l'état de l'objet doivent être privées.

Exemple:

```
#include <iostream>
#include <math.h>
using namespace std;
class complexe {
public:
     //constructeur
     complexe() {vide = true;}
```

```
      float norme() {
            if (vide == true) {
                  cout <<"norme() appelée sur objet non initialisé";
                  exit(-2);}
            else {
                  return sqrt (r*r + i*i);}
      }
      void set (float x,float y) {
            r = x; i = y;
            vide = false;
      }
      private : // protège les attributs
      float r,i;
      bool vide;
};
int main (){
      complexe y;
      y.r=1; // erreur de compilation
      y.i=1;// erreur de compilation
      cout << y.norme() << endl; // arrêt du programme
      return 0;
}
```

La solution est bien de définir les attributs i et r comme “private”.

3.2 Accès

Pour l’accès dans le programme aux parties publiques de la classe, la méthode est la même que pour les structures, pour une instance :

- `nom_variable.nom_attribut`,
- `nom_variable.nom_méthode`.

pour un pointeur sur une instance :

- `nom_variable->nom_attribut`,
- `nom_variable->nom_méthode`.

Dans l’implémentation de la classe, il peut être nécessaire d’appeler ou de manipuler des attributs ou des méthodes. Ceci se fait de manière implicite. Toutefois, il existe un membre caché créé automatiquement, appelé “`this`”. Ce dernier permet de faire référence à l’instance actuelle. En fait, lors de la déclaration d’une nouvelle variable (ou instance de la classe), le membre caché “this” reçoit l’adresse de l’instance. De manière générale, l’utilisation de “this” n’est pas utile.

Exemple:

```
class complexe {
... void set (float x, float y) {
      r=x; // ⟺ this− >r=x;
      i=y; // ⟺ this− >i=y;
      vide=false; // ⟺ this− >vide=false
} ...
```

3.3 Compilation séparée

L'opérateur de portée : " : :" est notamment utilisé quand la spécification et le corps des méthodes sont décrits dans des fichiers séparés (ce qui est vivement conseillé). **Exemple:**

complexe.h

```
#include <iostream>
#include <math.h>
using namespace std;
class complexe {
public:
     //constructeur
     complexe();      float norme();
     void set (float x,float y);
private : // protège les variables vide, r et i
     bool vide;
     float r,i; };
```

complexe.C

```
#include "complexe.h"
     complexe::complexe() {vide = true;}
     float complexe::norme() {
          if (vide == true) {
               cout <<"norme() appelée sur un objet non initialisé";
               exit(-2);}
          else {
               return sqrt (r*r + i*i);}
     }
     void complexe::set (float x,float y) {
          r = x; i = y;
          vide = false;
     }
```

Remarque:

Les fichiers sont nommés selon la norme suivante :

— fichier spécification : nom_fichier.h,

— fichier corps : nom_fichier.C ou nom_fichier.cc ou bien encore nom_fichier.cpp.

Ce type de déclaration a une conséquence sur l'expansion en ligne. Si le code de la méthode est donné dans la spécification de la classe, la méthode est automatiquement déclarée `inline`.

3.4 Constructeurs

Lors de la création d'une variable membre de la classe, trois actions sont réalisées automatiquement :

— l'association de la variable à un identificateur et à un type,

— l'allocation de l'espace mémoire nécessaire pour stocker la variable,

— l'initialisation éventuelle de la variable.

Pour les classes, on fait appel à un constructeur :

— Il peut en exister plusieurs pour une même classe,

— S'il n'y a pas de constructeur défini, la création de l'instance se fera automatiquement en créant toutes les données membres de cet objet,

— Le constructeur peut avoir éventuellement des paramètres,

— Si une classe possède plusieurs constructeurs, ils doivent avoir des arguments différents,

— Le constructeur qui peut être appelé sans argument est le constructeur pris par défaut,

— S'il n'y a pas de constructeur fourni dans la classe, C++ construit un constructeur par défaut ne prenant aucun paramètre et mettant à 0 les attributs et à NULL les pointeurs,

— Il existe un deuxième type de constructeur appelé : `constructeur de recopie`. La différence se fait au niveau des arguments. Ce dernier a un seul argument du type référence sur le type de la classe. Il existe toujours par défaut et il peut être redéfini par l'utilisateur,

— En C++, si une classe a un constructeur défini par l'utilisateur alors tous les objets de cette classe ne pourront être créés que par un constructeur défini par l'utilisateur ou par le constructeur de recopie.

Exemple:

perte de la définition automatique du constructeur sans argument

```
class complexe {
     public:
          ...
          complexe(float x,float y) {
               r=x; i=y;
               vide=false;
          }
          float norme() {
               ... suite inchangée
          };
int main () {
     complexe z(1,2); //OK, constructeur défini
     complexe y = z; //OK constructeur de recopie,
     // définition par défaut (champ par champ)
     complexe t(z); //OK constructeur de recopie,
     // définition par défaut.
     complexe x; /*ERREUR DE COMPILATION : le constructeur sans argument n'est plus défini par défaut car on définit complexe(float,float)*/
...
```

Voici un autre exemple d'une définition erronée du constructeur de recopie **Exemple:**

redéfinition erronée du constructeur de recopie

```
class complexe {
     public:
          ...
          complexe(complexe & autre) { // passage par référence obligatoire
               // vide = autre.vide; pour créer une erreur d'exécution
               r=autre.r;
               i=autre.i;
          }
          ...
};
int main () {
     complexe z;
     complexe t(z); //appel du constructeur de recopie,
     // défini ci-dessus
```

```
    t.norme(); // problème car on a oublié de
    //recopier "vide" dans la définition du constructeur de recopie.
    ...
}
```

Remarques:

- L'affectation et la recopie, par défaut, recopient récursivement champ par champ,
- Dans cet exemple (ou la copie est définie champ par champ), la redéfinition de l'opérateur de recopie est inutile,
- Il est permis dans la classe "complexe" d'accéder à la donnée privée "autre.vide",
- Pour créer un objet quelconque (pouvant contenir des données elles-mêmes instances de classe), le mécanisme est le suivant :
  - pour les types de base : allocation de l'espace nécessaire et initialisation à 0,
  - pour les autres types :
    - création de la partie héritée (vu dans la suite),
    - création des données membres dans l'ordre où elles sont définies,
  - s'il existe un constructeur défini, appel de ce constructeur,
- Il convient de bien différencier le constructeur de recopie de l'opérateur d'affectation :
  - `complexe x, z=x;` fait appel au constructeur de recopie `complexe(complexe & autre)`,
  - `complexe x, z; z=x ;` fait appel à l'opérateur d'affectation "=" car z est déjà défini,
- la meilleure définition d'un constructeur est celle qui permet de faire l'initialisation complète de l'objet uniquement avec des constructeurs. Dans ce cas, il faut préciser les constructeurs à choisir pour initialiser les attributs en les appelant explicitement. On utilise pour cela après la signature du constructeur la notation " :" suivi d'une liste de constructeurs à utiliser séparés par des virgules. Il se peut même que le corps du constructeur soit vide et se résume à une accolade ouvrante et fermante. **Exemple:**

```
class X {
// définition des attributs
     class Y yv;
     class Z zv;
     int i;
// définition du constructeur
     X(int v1, int v2, ...):yv(v1,v2,...),zv(..),i(v1){
...
}
```

Remarquez l'utilisation du constructeur de recopie des entiers appelé en écrivant : i(v1)

3.5 Destructeurs

Quand on sort d'un bloc du programme, les variables définies dans ce bloc sont détruites. Pour les classes, on fait appel à un destructeur. Le compilateur génère automatiquement le code du destructeur de l'objet. On peut toutefois créer son propre destructeur. Le nom de cette fonction membre particulière, pour un objet X sera ~X(). L'appel au destructeur est automatique à la fin du bloc où l'objet a été instancié. On ne doit jamais faire un appel explicite. Les destructeurs sont en général redéfinis quand, dans un constructeur, on a alloué de la mémoire en utilisant l'opérateur `new`. Il faut alors désallouer cette mémoire dans le destructeur car C++ ne possède pas de ramassage miettes.

```
class tab{
  float *t;
public:
  tab(int d){
      t=new float[d];
  }
  ~tab(){
    delete [] t;
  }
};
```