R2.01 : Développement Orienté Objets C++

François Delobel, Anaïs Durand, Abdel Hasbani, Laurent Provot, Carine Simon

IUT CLERMONT AUVERGNE
Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

Encapsulation

Protection et anti-intrusion

- Une personne est une entité de ce monde réel.
- Elle est autonome et :
  - Elle n'accepte pas qu'on vienne fouiller dans son portefeuille pour connaître son nom ou son âge.
    On doit le lui demander.
    Si elle veut bien, elle vous donne son nom ou son âge;
    sinon elle répond que ça ne vous regarde pas.
  - Vous ne pouvez pas changer son nom.
  - Tout renseignement la concernant doit lui être demandé.

Encapsulation

Effet Indésirable

```
class Client{
public :
    string nom, prenom;
    int anneeNaissance;
    double solde;

public :
    Client(string n, string p, int naiss, double unSolde=0);
    void crediter(double somme);
    double debiter(double somme);
    bool dansLeRouge();
    bool bonClient();
};
//--------------------
Client *dupont = new Client{"Dupont","Jean",1980,5000} ;
Dupont->solde = 0 ; // syntaxe correcte, mais il a tout perdu
                    // sans effectuer d'opération de débit.
delete dupont ;
```

Encapsulation

Droits d'accès

Il existe trois droits d'accès différents :

- public : l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
- private : l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet.
- Dans le cas de l'héritage, il existe en plus le modificateur protected (protégé).

Encapsulation

Principe

- L'encapsulation signifie qu'il n'est pas possible d'agir directement sur les données d'un objet
- Tout attribut d'instance ou de classe sera déclaré private
- 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

Dans notre exemple, si solde est privé $\Longrightarrow$ compilation refusée :

```
dupont->solde = 0 ;
```

error : 'double Client::solde' is private within this context

Encapsulation

Solution : getter/setter

- Accesseurs : accès en lecture (getter)

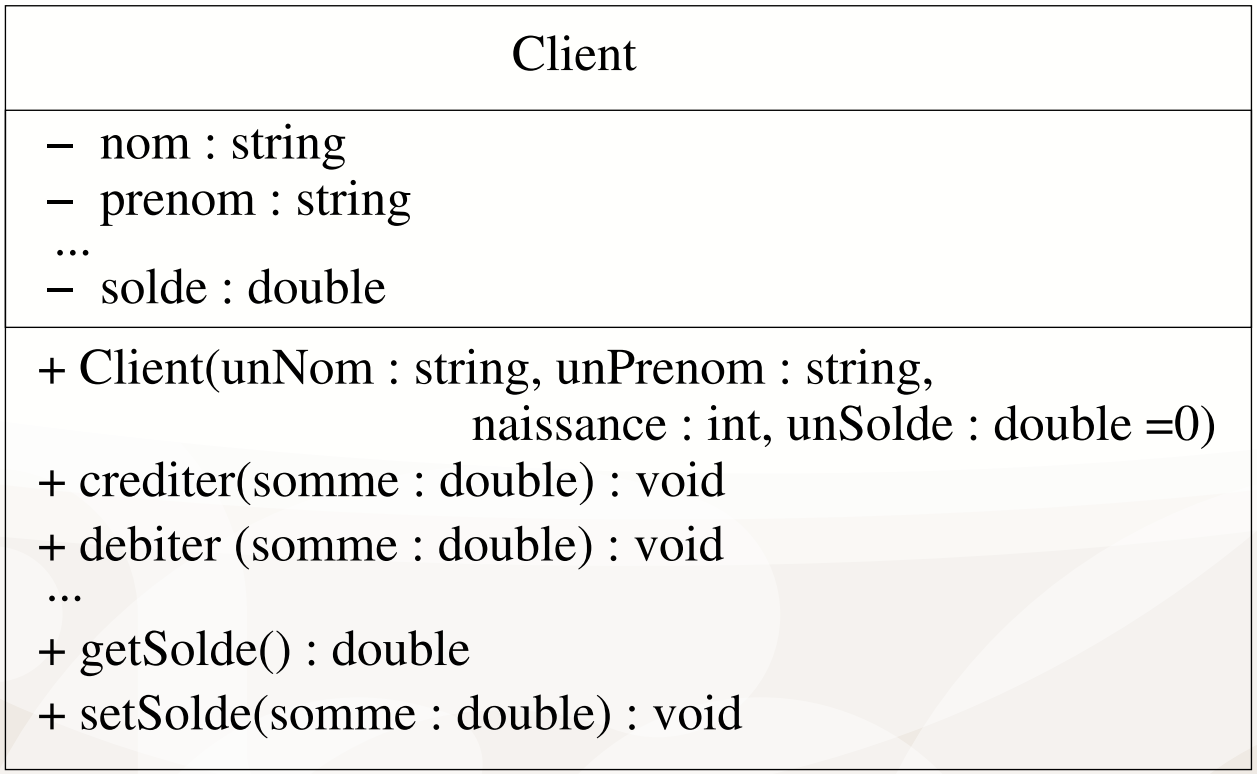
```
double Client::getSolde(){
    return solde ;
}
```

- Mutateur : accès en écriture (setter)

```
void Client::setSolde(double somme){
    solde = somme ;
}
```

Encapsulation

Représentation UML



Membres Statiques

Données Membres statiques

- Définition : un attribut statique est partagé par tous les objets de la même classe (une seule copie).
- Déclaration : avec le mot clé « static »
- Initialisation : à l'extérieur de la classe (dans le fichier.cpp)
- Accès : directement en utilisant l'opérateur de résolution de portée (::) pour spécifier sa classe.

Membres Statiques

Données Membres statiques

```
class Point{
private :
    double x, y;
    static int nbPoints;//  membre statique
    ...
};
int Point::nbPoints = 0; // initialisation d'un membre statique
Point::Point(double x, double y): x{x}, y{y}     // Constructeur
{
    nbPoints++;// un objet point de plus
}
...
Point::~Point(){ // Destructeur
    nbPoints--; // un objet point de moins
};
```

utilisation

```
cout<<"nombre de points créés : "<< Point::nbPoints << endl;
```

Membres Statiques

Fonctions membres statiques

- Définition : commune à tous les objets de la classe et qui existerait dès que la classe est définie
- Déclaration : avec le mot clé static
- Appel : en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (::)
- Particularité :
  - pas d'accès aux attributs de la classe
  - peut accéder aux données membres statiques

Membres Statiques

Fonctions membres statiques

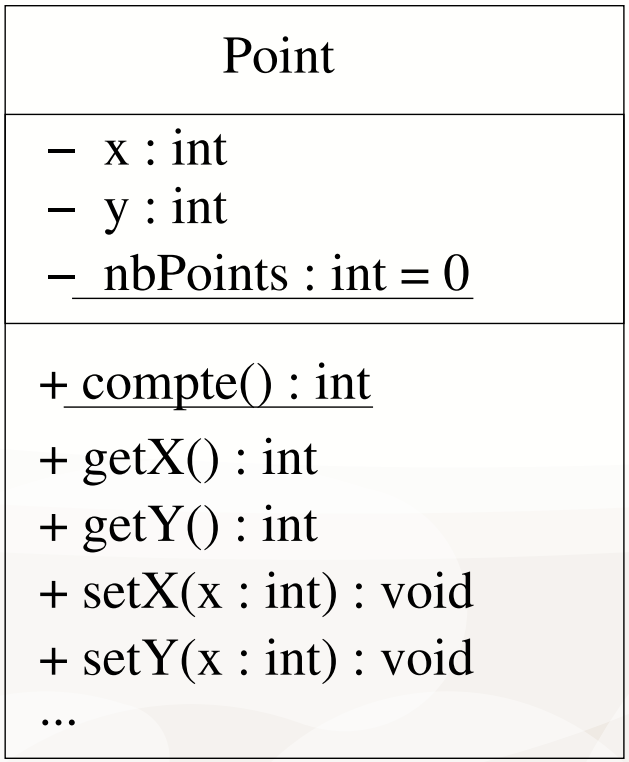
```
class Point{
private:
    double x, y ;
    static int nbPoints;  // attribut statique
    ...
public :
    static int compte(); // méthode statique qui retourne
                         // le nombre d'objets point
};
int Point::compte(){    //ici on ne met pas static
    return nbPoints;
}
```

Utilisation :

```
Point p0, p1{4, 0.0}, p2{2.5, 2.5};
Point *pp = new Point{1,1};
cout << "Il y a " << Point::compte() << " points\n";
// Affiche : Il y a 4 points
delete pp;
cout << "Il y a " << Point::compte() << " points\n";
// Affiche : Il y a 3 points
```

Membres Statiques

Représentation UML



Classes et fonctions amies

- On considère une classe A et une classe B
- En déclarant dans A, la classe B amie (de A), on permet à toutes les fonctions de la classe B d'accéder aux membres **privés** et **protégés** de la classe A.
- De la même manière, on peut déclarer une fonction F amie d'une classe qui permet à cette fonction (et seulement celle là) d'accéder aux données **privées** et **protégées** de la classe.

Classes et fonctions amies

Il existe plusieurs situations d'amitiés :

- fonction indépendante, amie d'une classe
- fonction membre d'une classe, amie d'une autre classe
- fonction amie de plusieurs classes
- toutes les fonctions membres d'une classe, amies d'une autre classe

Classes et fonctions amies

Exemple : fonction indépendante, amie d'une classe

Elle peut remplacer une fonction membre dissymétrique

```
class Point{
    public Point (int abs=0,int ord=0) : x{abs},y{ord}{ };
    friend bool coincide(const Point &, const Point &) ;
private :
    int x, y ;
};

bool coincide(const Point &p, const Point &q){
    return p.x == q.x && p.y == q.y ;
}

int main(){
    Point a{1,0}, b{1} ;
    if(coincide(a,b))  cout<<"a coincide avec b\n" ;
 // if(a.coincide(b))  cout<<"a coincide avec b\n" ;
}
```

Pointeur sur soi même (Auto référence)

Je suis un objet. Comment connaître mon adresse ?

```
class Monstre {
    /* ... */

    void afficherDescription() {
        cout << "mon adresse est..." << "?\n";
    }
};
```

Solution: mot-clef **this**

Pointeur sur soi même (Auto référence)

```
void Monstre::afficherDescription() {
    cout << "mon adresse est " << this << endl;
    cout << "ma force est de " << this->force << endl;
       // this est une adresse qu'on peut lire
}
void Monstre::setNom(string nom) {
    this-> nom = nom;
    // "this->" lève l'ambiguïté,
    //          différencie l'attribut de l'argument
}
void Monstre::attaquerFranchement(Chevalier *ch){
    parler(" prends ça " + ch->titre);
    ch->subirEtContreAttaquer(this);
    // passer sa propre adresse en paramètre
}
```

Les références en C++

L'opérateur & ci-dessous correspond à la définition d'une référence (où T est un type).

```
T var;
T & ref = var; // ref est une référence à var
```

Cela signifie que les identificateurs var et ref désignent la même variable.

Manipuler l'un, revient à manipuler l'autre.

Dès lors, var et ref sont des synonymes, ou des alias.

L'initialisation d'une référence lors de sa déclaration est obligatoire.

Les références en C++

```
int i=3;
int &refi = i;

printf("i = %d \n",i);
printf("refi = %d \n",refi);
printf("adresse de i = %x \n",&i);
printf("adresse de refi = %x \n",&refi);
printf("Incrémentation de i (i++)");
i++;
printf("refi = %d \n",refi);
```

Résultats :

```
i=3
refi = 3
adresse de i : 0065FDF4
adresse de refi : 0065FDF4
Incrément de i (i++) refi = 4
```

Les valeurs constantes

Le mot-clé **const** empêche toute modification ultérieure de la valeur d'une variable :

```
int const maVariable;
const int maVariable;
```

le mot-clé const s'applique à ce qui se trouve directement à sa gauche ou, s'il n'y a rien, à ce qui se trouve à sa droite

Les valeurs constantes

```
int i, j;
int const * p = &i; // *p devient une constante.
*p = j; //Erreur, la valeur pointée par p est constante.
p = &j; //Correct, le pointeur p n'est pas constant
int * const p; //Pointeur constant sur un int non constant
const int * const q; //Pointeur constant sur un int constant
```

```
char * const p=&c; // Affectation obligatoire
p++;   // Impossible
*p = 'r'; // Possible
```

Pointeur sur objet constant

```
const char *p;
p = &c; p++; // Possible
*p = 'R';   // Impossible
```

Les valeurs constantes

Remarques (concernant les références)

```
int &r ; // erreur, une référence doit être initialisée
int j ;
int &r{j} ; // ok r référence j
int &r1=r ; // ok r1 référence la même chose que r,
            //        donc référence aussi j
const int & cr{j} ; // une référence sur une constante peut
                    // être initialisée par une variable
const int c = 5 ;
int &cr2{c} ; // erreur, seule une référence sur
              //une cste peut référencer une cste.
const int &cr2{c} ; // Ok
```

Les méthodes constantes

Définition : méthodes qui ne modifient pas l'objet au travers duquel elles sont appelées.

```
class Image{
public:
    void affiche() const;
    ...
};
```

La méthode Image::affiche() ne pourra modifier aucun des attributs d'Image.

Seules les méthodes constantes peuvent être appelées par des objets constants.

Passage de paramètres en C++

- Par valeur (par défaut) : cas général
- Par valeur et constance des arguments :
  - cas des arguments muets constants
    la lvalue correspondante à l'argument ne doit pas être modifiée dans le corps de la fonction.
    Rq : Une lvalue désigne un objet avec une identité : il a une adresse définie et peut être accédé via un nom.
    Une rvalue désigne une valeur sans identité propre.

```
int n, *p; // n et p sont des lvalues
n = 1;p = &n; //1 et &n sont des rvalues
void f(const int nb){
    ...
    nb++ ; //rejeté en compilation
}
```

Passage de paramètres en C++

- Par adresse si besoin de modifier les arguments
- Transmission des arguments par référence Rappel : Une référence est un pointeur qui, une fois déclarée, s'écrit comme une simple variable

```
void echange(int &i, int &j) ;
int main(){
   int a=10 , b=20 ;
   cout << "- avant appel : a = " << a"<< b = " << b << endl;
   echange(a , b) ;
   cout << "- après appel : a = " << a"<< b = " << b << endl;
}
```

Passage de paramètres en C++

Transmission des arguments par référence

```
void echange(int &i, int &j){
    int c ;
    cout << "- début échange : " << i"<< " << j << endl ;
    c = i;
    i=j;
    j=c;
    cout << "- fin échange : " << i"<< " << j << endl ;
}
```

Résultat :

- avant appel : a = 10 b = 20
- début échange : 10 20
- fin échange : 20 10
- après appel : a = 20 b = 10

Passage de paramètres en C++

Transmission des arguments par référence

Une référence s'utilise pour :

- Modifier l'argument dans une fonction
- Désigner un objet qu'on ne possède pas
- Éviter la copie des arguments lors de l'appel de fonction
  - Gain de place dans la pile d'appel (mémoire)
  - Gain de temps CPU

Mais danger : Possibilité de modification des paramètres dans une fonction

Solution : utilisation des « const »

Passage de paramètres en C++

Transmission par référence d'une valeur de retour

```
int & f(){
    ...
    return n ;
}
```

À condition que n ne soit pas une variable locale.

- On pourra par exemple fournir en valeur de retour la référence à un objet créé dynamiquement ou renvoyer une référence qu'on aura reçue en argument.
- Conséquences : il devient possible d'utiliser son appel comme une lvalue modifiable : f(p) = 2*n + 5 ;

Passage de paramètres en C++

Transmission par référence d'une valeur de retour

```
int & alterne(int &n1, int &n2){
    static bool indice = true ; // variable static liée à alterne.
    if(indice){
        indice = false ;
        return n1 ;
    }
    else{
        indice = true ;
        return n2 ;
    }
}
int main(){
    int n=1, p=3, q=5 ;
    alterne(n, p) = 0 ; // on affecte 0 à la lvalue dont
                        //   alterne fournit la référence
    cout << "n = " << n << ", p = " << p << endl ;
    alterne(p, q) = 12 ;
    cout << "p = " << p << ", q = " << q << endl ;
}
Résultats :   n = 0,  p = 3
              p = 3,  q = 12
```