Rappels de C++

Programmation orientée objet

Vincent Lemaire
vincent.lemaire@upmc.fr

Sur le type **`class`**/**`struct`**

- Déclaration, utilisation d'une **`class`**/**`struct`**
- Constructeurs, destructeurs
- **`const`**, **`friend`**, ...
- Exemple complet

Héritage (simple)

- Héritage
- Membres protégés, règles d'héritage

Un premier mot sur le polymorphisme dynamique

- Type statique, type dynamique
- Méthode virtuelle, classe abstraite

Déclaration d'une `class`/`struct`

Les types **`struct`** et **`class`** permettent de regrouper des :

- variables membres
- fonction membres (méthodes)

Ces membres peuvent être déclarés comme

- **`public`** : accessibles dans tout le programme
- **`private`** : accessibles uniquement par les méthodes de la classe (et aussi...)

Par défaut les membres sont

- **`public`** pour le type **`struct`**
- **`private`** pour le type **`class`**

Les méthodes peuvent être définies

- à l'intérieur de la définition de la **`class`**/**`struct`**
- à l'extérieur avec l'opérateur de résolution de portée **`::`**

Exemple de déclaration d'une classe

On reprend l'exemple des nombres p-adiques

```
class p_adic {
    public:
        p_adic(unsigned p, unsigned n);
        p_adic(unsigned p, unsigned r, unsigned a[]);
        void change_base(unsigned p);
        void affiche();
    private:
        unsigned n;
        unsigned p, r;
        unsigned * coeff;
        bool verif();
};
```

La définition des méthodes se fera plus loin.

Il y a 3 méthodes (`change_base`, `affiche` et `verif`) et 2 constructeurs.
Un constructeur est une fonction membre :

- qui a le même nom que la **`class`**
- qui ne renvoie rien (pas de **`return`**, pas de type de sortie)
- qui sert à initialiser les variables membres

Accès aux membres

Une variable de type **class** est appelée objet.
A la création d'un objet, un constructeur est obligatoirement appelé.

```
unsigned a[] = { 4, 1 };
p_adic X(5, 2, a);
p_adic * Y = new p_adic(10);
```

Comme le type **struct** du langage C, on peut accéder aux membres

- d'un objet avec l'opérateur `.`
- d'un objet pointé avec l'opérateur `->` (remplace l'utilisation conjointe des opérateurs `*` et `.`)

```
X.affiche();
Y->affiche(); // ou (*X).affiche();
```

Dans toutes les méthodes, il y a une définition implicite du pointeur **this** qui pointe sur l'objet courant (l'objet de la méthode appelée).

Constructeur par défaut

- constructeur sans arguments (ou tous ayant une valeur par défaut)
- le constructeur appelé à la création d'un objet s'il n'y a pas d'appel explicite
- synthétisé par le compilateur si aucun constructeur n'est défini ATTENTION (exemple au tableau)

La classe `p_adic` n'a pas de constructeur par défaut. Impossible d'écrire

```
p_adic Z;
p_adic tab[10];
```

Pour créer un constructeur par défaut (ce qui est recommandé) on peut mettre des valeurs par défaut à un constructeur déjà défini :

```
class p_adic {
public:
    p_adic(unsigned p = 2, unsigned n = 0);
...
}
```

Constructeur par copie

- constructeur d'une classe `C` prenant comme unique argument : une référence sur un `C` constant : « `C const &` » ou « `const C &` »
- synthétisé par le compilateur s'il n'est pas défini
  ATTENTION, copie « bit à bit »

La classe `p_adic` n'a pas de constructeur par copie, une version est donc créée par le compilateur et l'on peut écrire

```
p_adic Z = p_adic(X);
p_adic U = X;
```

La version synthétisée est erronée car les objets `X` et `Z` (ou `U`) vont partager le même pointeur `coeff` : si `X` est détruit avant `Z`, alors le pointeur `coeff` de `Z` devient invalide (et dangereux).

Liste d'initialisation

- syntaxe spéciale pour un constructeur qui permet d'initialiser les variables membres (notamment les objets membres n'ayant pas de constructeur par défaut)

```
class p_adic {
public:
    p_adic(unsigned p = 2, unsigned n = 0); // défini en dehors
    p_adic(unsigned p, unsigned r, unsigned a[])
        : p(p), r(r), coeff(new unsigned[r]), n(0) {
        double pk = 1;
        for (int k = 0; k < r; k++) {
            coeff[k] = a[k];
            n += a[k] * pk;
            pk *= p;
        }
    }
    ...
}
```

Destructeur

Fonction membre spéciale (unique)

- même nom que la classe précédé du caractère ~
- ne prend pas d'arguments, ne renvoie rien
- est appelé lorsqu'un objet est détruit
- synthétisé par le compilateur s'il n'est pas défini
  ATTENTION

Encore une fois la version créée par le compilateur est erronée, il faut le définir !

```
class p_adic {
public:
    p_adic(unsigned p = 2, unsigned n = 0); // défini en dehors
    ...
    ~p_adic() {
        delete [] coeff;
    }
    ...
}
```

De l'usage des `const`

Il est fondamental de bien utiliser le mot-clé **`const`**.
Peuvent être déclarées comme constantes :

- variables membres : syntaxe habituelle, signification habituelle
- fonctions membres : ajout de **`const`** juste après le nom suivi des arguments
  - le mot-clé **`const`** fait partie de la signature de la fonction
  - permet de préciser que la fonction membre ne modifie pas l'objet
  - améliore la portée de la fonction

Les fonction `affiche` et `verif` devraient être qualifiées **`const`**.

```
class p_adic {
public:
    ...
    void change_base(unsigned p);
    void affiche() const;
private:
    ...
    bool verif() const;
};
```

Accesseurs / mutateurs : exemple au tableau.

Fonctions et classes amies

Fonction amie : d'une classe `C`

- peut être une fonction (globale) ou une fonction membre d'une classe `D`
- doit être déclarée dans la classe `C` précédée du mot-clé **`friend`**
- a le droit d'accéder à tous les membres de la classe `C`

Si on écrit la fonction `affiche` comme fonction `amie`

```
class p_adic {
public:
    ...
    void change_base(unsigned p);
    friend void affiche(p_adic const &);
private:
    ...
    bool verif() const;
};
```

Classe amie : d'une classe `C`

- doit être déclarée dans la classe `C` précédée du mot-clé **`friend`**
- a le droit d'accéder à tous les membres de la classe `C`

ATTENTION : relation non transitive !
les amis de mes amis ne sont pas mes amis...

Exemple complet, déclaration dans le header

```
class p_adic {
public:
    p_adic(unsigned p = 2, unsigned n = 0);
    p_adic(unsigned p, unsigned r, unsigned a[])
        : p(p), r(r), coeff(new unsigned[r]), n(0) {
        // code du constructeur
    }
    ~p_adic() { delete [] coeff; }

    // méthodes publiques
    void change_base(unsigned p);
    void affiche() const;

    // fonctions amies
    friend unsigned je_suis_amie(p_adic const &);
private:
    unsigned n;
    unsigned p, r;
    unsigned * coeff;
    bool verif() const;
};
```

Exemple complet, définitions dans le source

```
p_adic::p_adic(unsigned p, unsigned n) {
// code du constructeur
};

void p_adic::change_base(unsigned p) {
// code...
};

void p_adic::affiche() const {
// code...
};

bool p_adic::verif() const {
// code...
};

unsigned je_suis_amie(p_adic const &X) {
// code...
};
```

Héritage

L'héritage est une relation

- transitive
- non symétrique
- non réflexive
- non cyclique

Si une classe `B` hérite d'une classe `A` on dira de façon équivalente :

- `B` dérive de `A`
- `B` est la classe fille de `A`
- `A` est la classe mère de `B`

La syntaxe pour l'héritage publique est la suivante :

```
class B : public A {
// définition de la classe B
}
```

Héritage -2-

Moralement, un objet de classe `B` « contient » un objet de la classe mère `A`.

- le constructeur de `B` doit appeler explicitement celui de `A`
- une méthode de `B` de même signature qu'une méthode de `A` masque cette dernière
- un objet de `B` est accepté partout où un objet de `A` est attendu
  l'objet est tronqué !
- l'adresse d'un objet de `B` est compatible avec celle d'un objet de `A` :
  propriété essentielle de l'héritage !

Exemple fil rouge : un `Pixel` dérive d'un `Point`

```
class Point {
public:
    Point(int x = 0, int y = 0) : x(x), y(y) {};
    void affiche() const {
        std::cout << x << "\t" << y; }
private:
    int x, y;
}
```

Héritage -3-

```
class Pixel : public Point {
public:
    Pixel(int x = 0, int y = 0, std::string col = "")
        : Point(x, y), col(col) {};
private:
    std::string col;
}
```

Exemple d'utilisation :

```
Point P(2, 3);
Pixel Q(2, 3, "vert");
P.affiche();
Q.affiche(); // OK
```

Héritage -3-

```
class Pixel : public Point {
public:
    Pixel(int x = 0, int y = 0, std::string col = "")
        : Point(x, y), col(col) {};
private:
    std::string col;
}
```

Exemple d'utilisation :

```
Point P(2, 3);
Pixel Q(2, 3, "vert");
P.affiche();
Q.affiche(); // OK
```

Peut-on redéfinir la fonction `affiche` dans la classe `Pixel` de la façon suivante ?

```
class Pixel : public Point {
    ...
    void affiche() const {
        std::cout << x << "\t" << y << "\t" << col; }
    ...
}
```

Membres protégés

NON ! Les variables `x` et `y` ne sont pas membres de la classe `Pixel`.

Les règles d'accessibilité des membres pour l'héritage publique sont :

- les membres **`public`** de la classe mère sont accessibles à la classe fille
- les membres **`private`** de la classe mère sont inaccessibles à la classe fille

Pour pallier à cette rigidité trop grande, il existe des membres protégés que l'on déclare avec le mot-clé **`protected`** :

- les membres **`protected`** de la classe mère sont accessibles à la classe fille (et ses amies) mais inaccesible au reste du programme (comme **`private`**).

Membres protégés

NON ! Les variables x et y ne sont pas membres de la classe `Pixel`.

Les règles d'accessibilité des membres pour l'héritage publique sont :

- les membres **`public`** de la classe mère sont accessibles à la classe fille
- les membres **`private`** de la classe mère sont inaccessibles à la classe fille

Pour pallier à cette rigidité trop grande, il existe des membres protégés que l'on déclare avec le mot-clé **`protected`** :

- les membres **`protected`** de la classe mère sont accessibles à la classe fille (et ses amies) mais inaccesible au reste du programme (comme **`private`**).

En déclarant x et y comme **`protected`** dans `Point` le code précédent est valide mais il est plus élégant d'écrire :

```
class Pixel : public Point {
    ...
    void affiche() const {
        Point::affiche();
        std::cout << "\t" << col; }
    ...
}
```

Héritage privé, publique

Par défaut l'héritage est privé, la syntaxe est

```
class B : A {
// definition de B
};
```

Règles héritage privé :

- **public**, **protected** devient **private**
- **private**, inaccessible devient inaccessible

Règles héritage publique :

- **public** reste **public**
- **protected** reste **protected**
- **private**, inaccessible devient inaccessible

Il existe aussi l'héritage protégé...

Type statique

Quels sont les types des variables suivantes ?

```
Pixel Px(2, 3, "vert");
Point Pt = Px;
Point * y = & Px;
Point & r = Px;
```

Sur quoi pointe y ?

Sur quoi réfère r ?

Que donne le code suivant ?

```
Px.affiche();
Pt.affiche();
y->affiche();
r.affiche();
```

Type statique

Quels sont les types des variables suivantes ?

```
Pixel Px(2, 3, "vert");
Point Pt = Px;
Point * y = & Px;
Point & r = Px;
```

Sur quoi pointe y ?
Sur quoi réfère r ?

Que donne le code suivant ?

```
Px.affiche();
Pt.affiche();
y->affiche();
r.affiche();
```

Le mécanisme d'appel d'une méthode est lié au type statique.
Les fonctions appelées seront

```
y->Point::affiche();
r.Point::affiche();
```

Type dynamique

Le type statique se lit dans le code source : il est déterminé à la compilation.
Le type dynamique est lui déterminé à l'execution.

Dans l'exemple précédent

- `y` est initialisé avec l'adresse du `Pixel Px` donc le type dynamique est : pointeur sur `Pixel`
- `r` est initialisé avec le `Pixel Px` donc le type dynamique est : référence sur `Pixel`

Type dynamique

Le type statique se lit dans le code source : il est déterminé à la compilation.
Le type dynamique est lui déterminé à l'execution.

Dans l'exemple précédent

- y est initialisé avec l'adresse du `Pixel Px` donc le type dynamique est : pointeur sur `Pixel`
- `r` est initialisé avec le `Pixel Px` donc le type dynamique est : référence sur `Pixel`

La fonction **`typeid`** permet d'identifier le type d'une variable lors de l'execution du programme.

- nécessite l'en-tête `typeinfo`
- renvoie un objet `typeinfo` qui possède des opérateurs de comparaison et une fonction membre `name` (qui renvoie une chaîne de caractère)
- point technique pour le compilateur `g++` on doit inclure l'en-tête `cxxabi.h` et utiliser la fonction `abi::__cxa_demangle`

Exercice : `typeid`

En utilisant la fonction suivante

```
void affiche_type(const std::type_info &ti) {
int status;
cout << abi::__cxa_demangle(ti.name(), 0, 0, &status) << endl;
};
```

Coder et exécuter le programme suivant

```
Pixel Px(2, 3, "vert");
affiche_type(typeid(Px));
affiche_type(typeid(&Px));

Point &r = Px;
affiche_type(typeid(r));
affiche_type(typeid(&r));

Point *y = &Px;
affiche_type(typeid(*y));
affiche_type(typeid(y));
```

Méthode virtuelle, classe abstraite

Retour à notre exemple...

```
Pixel Px(2, 3, "vert");
Point * y = & Px;
Point & r = Px;
y->affiche();
r.affiche();
```

On souhaite que les appels `y->affiche()` et `r.affiche()` exécutent la méthode `affiche()` du `Pixel Px`.

Une méthode devant être spécialisée dans ses classes filles doit être qualifiée de virtuelle :

- le mot-clé **`virtual`** précède le prototype de la méthode (dans la classe mère)

Le mécanisme d'appel d'une méthode virtuelle est lié au type dynamique.

Méthode virtuelle, classe abstraite

```
class Point {
public:
    Point(int x = 0, int y = 0) : x(x), y(y) {};
    virtual void affiche() const {
        std::cout << x << "\t" << y; }
protected:
    int x, y;
}

class Pixel : public Point {
public:
    Pixel(int x = 0, int y = 0, std::string col): Point(x,y), col(col){}
    void affiche() const {
        Point::affiche();
        std::cout << "\t" << col; }
private:
    std::string col;
}
```

Méthode virtuelle pure et classe abstraite

Méthode virtuelle pure :

- méthode virtuelle qui n'a de sens que pour les classes dérivées (et non pour la classe mère)
- syntaxe : mot-clé **virtual** devant le prototype et = 0 après le prototype

```
class VarAlea {
    public:
        virtual double density(double x) = 0;
    protected:
        ...
}
```

Classe abstraite :

- classe qui contient au moins une méthode virtuelle pure
- il ne peux pas exister d'objets de cette classe