

Le type class

Introduction au C++ et à la programmation objet

E. Courcelle

CALMIP, UMS 3669

Décembre 2019

Types et objets

ATTENTION, Ne pas confondre

Une **classe** est un **type**. Un **objet** est une **variable**. Une **classe** permet d'instancier une variable, comme un **moule** permet de faire un objet en plastique.

Une déclaration de classe

```
class complexe {  
public:  
    void init(float x, float y);  
    void copie(const complexe& y);  
private:  
    float r;  
    float i;  
}
```

On trouve quoi dans une classe ?

- Des déclarations de types (types locaux)
- Des variables (membres, propriétés, ...)
- Des déclarations de fonctions (fonctions membres, méthodes)
- Des définitions de fonctions inline (plus rapide si la fonction est triviale)

```
class complexe {  
public:  
    void init(float x, float y);  
    void copie(const complexe& y);  
private:  
    float r;  
    float i;  
}
```

Section private

Utilisable **presque** uniquement à partir d'une **variable de la même classe**.
Donc Le code suivant ne compile pas:

```
complexe x;  
...  
x.r = 0.0;  
x.i = 0.0;
```

Par contre, le code suivant fonctionne (Attention aux notations !):

```
class complexe {  
public:  
    void init(float x, float y) {r=x; i=y;};  
    void copie(const complexe& y) {r=y.r; i=y.i;};  
private:  
    float r;  
    float i;  
}
```

Section public

Utilisable depuis partout, y compris le code extérieur à l'objet.

Donc Le code suivant fonctionne:

```
complexe x;  
...  
x.init(0,0);
```

Section protected

Utilisable depuis les classes dérivées uniquement, cf. plus tard !

Une classe peut choisir ses amis

Une classe peut déclarer explicitement que certaines fonctions, ou certaines autres classes, sont **amies**.

Dans ce cas, les fonctions, ou les fonctions membres des classes amies **ont accès à la section privée de la classe en question**.

```
class complexe {
public:
    void init(float x, float y) {r=x; i=y;};
    void copie(const complexe& y) {r=y.r; i=y.i;};
    friend complexe addition(const complexe &c1, const complexe& c2);

private:
    float r;
    float i;
}

complexe addition(const complexe &c1, const complexe& c2) {
    complexe c3;
    c3.r = c1.r + c2.r;
    c3.i = c1.i + c2.i;
    return c3;
}
```

Voir également plus loin.

On met quoi où ?

Dans la section private

Tout ce qui se rapporte au **fonctionnement interne de l'objet**: variables et fonctions.

Dans la section public

- Toutes les fonctions d'interface mais **pas de variables**.
- On **change** la valeur d'une propriété par une fonction **mutante** (mutator)
- On **lit** la valeur d'une propriété par une fonction de type **accesseur** (accessor)

Calcul du module intégré à la classe

Une nouvelle version de complexe, avec calcul du module:

```
class complexe {
public:
    void init(float x, float y) {r=x; i=y; _calc_module();};
    copie(const complexe& y) {r=y.r; i=y.i; m=y.m;};
    float get_r() { return r;};
    float get_i() { return i;};
    void set_r(float x) { r=x; _calc_module();};
    void set_i(float x) { i=x; _calc_module();};
    float get_m() {return m;};
private:
    float r;
    float i;
    float m;
    void _calc_module();
}

void complexe::_calc_module() {
    m = sqrt(r*r + i*i);
}
```

m (module), variable privée

Le membre m est privé, donc on ne pourra pas écrire (**et c'est dommage !**):

```
complexe c ;  
...  
cout << c.m; // Ne compile pas
```

On devra écrire (un peu plus lourd, **mais pas plus lent**):

```
complexe c ;  
...  
cout << c.get_m() ;
```

On ne pourra pas non plus écrire (**et c'est tant mieux !**):

```
complexe c ;  
...  
c.m = 56; // Ne compile pas
```

L'objet reste toujours cohérent

Une version avec calcul du module au fil de l'eau

Une nouvelle version de complexe, optimisée pour les fonctions mutantes.

```
class complexe {  
public:  
    void init(float x, float y) {r=x; i=y;};  
    copie(const complexe& y) {r=y.r; i=y.i;};  
    float get_r() { return r;};  
    float get_i() { return i;};  
    void set_r(float x) { r=x;};  
    void set_i(float x) { i=x;};  
    float get_m() {return sqrt(r*r+i*i);};  
private:  
    float r;  
    float i;  
}
```

Vu depuis l'extérieur de l'objet, rien n'a changé

Le meilleur des deux mondes

Une nouvelle version de complexe, optimisée pour tous les cas d'utilisation.

```
class complexe {
public:
    void init(float x, float y) {r=x; i=y; m=0; m_flg=false;};
    void copie(const complexe& y) {r=y.r; i=y.i; m=y.m;};
    float get_r() { return r;};
    float get_i() { return i;};
    void set_r(float x) { r=x; m_flg=false;};
    void set_i(float x) { i=x; m_flg=false;};
    float get_m();
private:
    float r;
    float i;
    bool m_flg;
    float m;
    void _calc_module() {m=sqrt(r*r+i*i);};
};

float complexe::get_m() {
    if (!m_flg) {
        _calc_module();
        m_flg=true;
    };
    return m;
};
```

Vu depuis l'extérieur de l'objet, rien n'a changé

Ne pas abuser des accesseurs !

Dans le code précédent, il n'y a pas de fonction getmflg
Pourquoi ?

Définition du constructeur

Encore un nouveau complexe:

```
class complexe {
public:
    complexe( float x, float y): r(x), i(y), m_flg( true ), m( _calc_module() )
    {};
    void copie( const complexe& y ) { r=y.r; i=y.i; m=y.m; };
    float get_r() { return r; };
    float get_i() { return i; };
    void set_r( float x ) { r=x; m_flg=false; };
    void set_i( float x ) { i=x; m_flg=false; };
    float get_m();
private:
    float r;
    float i;
    bool m_flg;
    float m;
    void _calc_module() { m=sqrt( r*r+i*i ); };
};
```

Le constructeur:

- A un nom imposé
- Renvoie le nouvel objet construit mais **on ne le dit pas !**

Redéfinition du constructeur

Le constructeur a été redéfini, et maintenant nous pouvons écrire:

```
complexe x(2,0);
```

Nous pouvons aussi écrire (**en C++11**):

```
complexe x = { 2, 0 };
```

Mais nous ne **pouvons pas** écrire:

```
complexe x;
```

car le constructeur prédéfini (fourni par le compilateur) **a disparu** !

La liste d'initialisation

Le constructeur est The place to be pour:

- Initialiser les membres
- Demander des ressources (mémoire, fichiers, etc.)

Initialiser les variables

- Cela se fait dans la liste d'initialisation
- Placée et exécutée **avant** le code du constructeur.
- Peut être vide (si rien à initialiser)

Demander des ressources

- Cela se fait dans le corps du constructeur
- Placé et exécuté **après** l'initialisation des variables.
- Peut être vide (si pas de ressource à demander)

```
classe complexe {  
    public:  
        complexe(float x, float y): r(x), i(y) {};  
}
```


Le destructeur

Le destructeur:

- A un nom imposé: `~` complexe
- Ne renvoie rien
- est The place to be pour rendre au système les ressources qu'il vous a prêtées

Variables et fonctions-membres statiques

Le code suivant permet d'avoir un mode "debug" à la classe complexe. nous allons positionner une variable booléenne, la valeur de cette variable est **commune à tous les objets de la classe**.

```
class complexe {
public:
    ...
    static void set_debug() { debflg=true; };
    static void clr_debug() { debflg=false; };
private:
    ...
    static bool debflg;
    ...
};

// Allocation memoire et initialisation dans le programme principal
bool complexe::debflg=false;

main () {
    complexe::set_debug();           // passe en mode debug
    ...
    complexe::clr_debug();           // sort du mode debug
}
```

Variables et fonctions-membres statiques (2)

ATTENTION, PIEGE !

Une donnée membre statique ressemble beaucoup à une variable globale:

- L'emplacement mémoire doit être alloué dans le programme principal, pas par l'appel au constructeur
- Sa durée de vie s'étend sur toute la durée du programme, indépendamment des instances des objets
- Par contre, elle obéit aux **mêmes règles de portée** que les autres données membres de la classe.

Membres constants

Le code suivant encapsule un tableau C:

```
class tableau {
public:
    tableau(int);
    ~tableau() {free(buffer); buffer = NULL;};
private:
    const size_t taille;
    int* buffer;
};

tableau::tableau(int s) : taille(s) {
    buffer = (int *) malloc ( taille * sizeof(int) );
};

void main() {
    tableau t1(1000);
    tableau t2(10000);
};
```

- La taille du tableau est choisie à la construction de l'objet
- La taille du tableau, un fois construit, ne **peut pas changer** par la suite
- Le membre taille est donc déclaré comme constant

Objets constants

Avec la classe complexe précédemment écrite, le code suivant ne compile pas:

```
const complexe i(0,1);  
float X = i.get_i(); // Qui a dit que get_i est un accesseur ?
```

Il suffit de réécrire la classe pour exprimer le fait que les accesseurs laissent l'objet constant:

```
class complexe {  
private:  
    float r;  
    float i;  
    ...  
public:  
    complexe(float , float);  
    float get_r() const { return r;};  
    float get_i() const { return i;};  
    ...  
};
```

Membres mutable

Nous avons un souci avec le complexe version 3 (le plus performant !). Le code suivant ne compilera pas, car si c'est un accesseur du point-de-vue **logique**, en réalité il **modifie l'objet**:

```
float complexe::get_m() const {  
    if (!m_flg) {  
        _calc_module();  
        m_flg=true;  
    };  
    return m;  
};
```

Pour s'en sortir, on déclare certains membres **mutable**:

```
class complexe {  
public:  
    ...  
    float get_r() const { return r;};  
    float get_i() const { return i;};  
    float get_m() const;  
private:  
    ...  
    mutable bool m_flg;  
    mutable float m;  
    void _calc_module() const {m=sqrt(r*r+i*i);};  
};  
  
float complexe::get_m() const {  
    ...  
}
```

This, c'est tout moi

Tout code d'une fonction-membre peut utiliser le pointeur **this**, qui pointe sur l'objet courant. Cela est utile pour enchaîner les fonctions set dans le cas du complexe:

```
class complexe {
public:
    complexe& set_r(float x) { r=x; return *this; };
    complexe& set_i(float y) { r=y; return *this; };
private:
    ...
}
complexe x(0,1);
x.set_r(1).set_r(0);
```

Opérateurs

Cette structure est très utile dans la redéfinition des opérateurs, car c'est elle qui permettra d'écrire du code de la même forme avec des variables de types de base et avec des objets.

(à suivre)