C++ - Classes et objets

UE 15 - Informatique appliquée

F. Pluquet

HelHa
Slides originaux de R. Absil (ESI)

14 octobre 2018

Table des matières

Classes

Les structures et les classes

- Les structures et les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++ (pas en C), on peut définir des fonctions membres dans des structures
  - Méthodes

Différences entre classes et structures

- En l'absence de spécificateur d'accès,
  1. les membres d'une classe sont privés, ceux d'une structure sont publics
  2. les membres d'une mère au sein d'une fille sont privés dans une classe, publics dans une structure

Exemple

- Fichier `point_struct.cpp`

```
1  struct Point
2  {
3    Point(int x, int y)
4    {
5      this->x = x;
6      this->y = y;
7    }
8
9    inline double getX()
10   {
11     return x;
12   }
13
14   inline double getY()
15   {
16     return y;
17   }
18
19   double dist(Point p)
20   {
21     return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
22   }
23
24   private:
25     double x, y;
26 };
```

Exemple

- Fichier `point_class.cpp`

```
1  class Point
2  {
3    double x, y;
4
5    public:
6      Point(int x, int y)
7      {
8        this->x = x;
9        this->y = y;
10     }
11
12     inline double getX()
13     {
14       return x;
15     }
16
17     inline double getY()
18     {
19       return y;
20     }
21
22     double dist(point p)
23     {
24       return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
25     }
26 };
```

Utilisation

- Fichiers `point_class.cpp` et `point_struct.cpp`

```
1  int main()
2  {
3    Point p1(1,1);
4    //cout << p1.x << " " << p1.y << endl; //ko
5    cout << p1.getX() << " " << p1.getY() << endl;
6    point p2(2,2);
7    cout << p2.getX() << " " << p2.getY() << endl;
8    cout << "dist = " << p1.dist(p2) << endl;
9  }
```

Remarque

- Ne pas oublier le ';' après la déclaration d'une classe ou d'une structure

Constructeurs et destructeurs

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

Constructeur

- Fonction particulière appelée à l'instanciation
  - Initialisation, copie ou réallocation
  - Pas à l'affectation
- Pas de type de retour
- Même nom que la classe
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Si aucun constructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- Appelé implicitement à la déclaration sans paramètres

Constructeur de recopie

- Appelé implicitement quand un paramètre est passé par valeur
  - Copie implicite effectuée
- Constructeur avec un paramètre constant passé par référence
  - Paramètre de même type que la classe
- Si aucun constructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Copie toujours possible

Destructeur

- Fonction particulière appelée à la destruction de l'objet
  - Désallocation implicite ou explicite
  - Explicite via `delete`
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de ~
- Unique
  - Si aucun : comportement par défaut
  - Si plusieurs déclarés : erreur
- Règles d'appel particulières en cas d'héritage

Exemple

- Fichier `point-cstr.cpp`

```
1  Point::Point(int x = 0, int y = 0)
2  {
3    this->x = x; this->y = y;
4    copie = false;
5    cout << "Construction de " << x << " " << y << endl;
6  }
7
8  Point::Point(const Point& p)
9  {
10   this-> x = p.x; this-> y = p.y;
11   copie = true;
12   cout << "Copie de " << x << " " << y << endl;
13 }
14
15 Point::~Point()
16 {
17   cout << "Destruction de " << x << " " << y;
18   if(copie)
19     cout << " (copie)";
20   cout << endl;
21 }
22
23 void sayHello(const Point& p)
24 {
25   cout << "Hello Mr Point " << p.getX() << " " << p.getY() << endl;
26 }
```

Exemple (2/2)

- Fichier `point-cstr.cpp`

```
1  int main()
2  {
3    Point p1; Point p2(1,1);
4    cout << p1.getX() << " " << p1.getY() << endl;
5    sayHello(p1);
6    cout << p2.getX() << " " << p2.getY() << endl;
7    cout << "dist = " << p1.dist(p2) << endl;
8    Point p3(p1);//explicit copy
9    p3 = p2;
10 }
```

Remarques

- Copies implicites effectuées
- Destructions implicites effectuées

Liste d'initialisation

Principe

- Permet d'initialiser à la volée les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs sans constructeur par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Exemple

- Fichier `point_init.cpp`

```
1  class Point
2  {
3    double x, y;
4
5    public:
6      Point(int x = 0, int y = 0) : x(x), y(y) {}
7
8      double getX() const
9      {
10       return x;
11     }
12
13     double getY() const
14     {
15       return y;
16     }
17
18     double dist(Point p)
19     {
20       return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
21     }
22 };
```

Exemple

- Fichier `deleg.cpp`

```
1  class A
2  {
3    int i;
4    const int k;
5
6    private:
7      A() : k(5)
8      {
9        cout << "Init " << endl;
10       //k = 5;
11     }
12
13   public:
14     A(int x) : A()/*, i(x)*/
15     {
16       i = x;
17     }
18
19     void print() { cout << "A : " << i << endl;}
20 };
```

Absence de constructeur par défaut

- Fichier `no-cstr.cpp`

```
1  struct A
2  {
3    int i;
4    A(int i) : i(i) {}
5  };
6
7  struct B
8  {
9    A a;
10   B(A a) : a(a) {};//ok
11
12   //B(A a)//ko
13   //{
14   //  this -> a = a;
15   //}
16 };
17
18 int main()
19 {
20   A a(2);
21   B b(a);
22 }
```

Fonctions membres constantes

- Usage du mot-clé `const` à la fin du prototype
- Ne modifie pas `this`
- Modification impossible d'un attribut
- Appel impossible d'une fonction non constante sur `this`
- Souvent utilisé pour les getters
- Si un objet est `const`, on ne peut appeler que des fonctions `const` dessus

Inclusions de fichiers

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.cpp`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée

Exemple

- Fichiers `point_decl.h`, `point_decl.cpp` et `point_decl-main.cpp`

```
1  class Point
2  {
3    double x, y;
4
5    public:
6      Point(double x, double y);
7      inline double getX() const;
8      inline double getY() const;
9      double dist(Point p) const;
10 };
```

```
1  int main()
2  {
3    Point p1(1,1);
4    //cout << p1.x << " " << p1.y << endl; //ko
5    cout << p1.getX() << " " << p1.getY() << endl;
6    Point p2(2,2);
7    cout << p2.getX() << " " << p2.getY() << endl;
8    cout << "dist = " << p1.dist(p2) << endl;
9  }
```

Exemple

- Fichiers `point_decl.h`, `point_decl.cpp` et `point_decl-main.cpp`

```
1  double Point::getX() const
2  {
3    return x;
4  }
5
6  double Point::getY() const
7  {
8    return y;
9  }
```

```
1  #include "point_decl.h"
2
3  Point::Point(double x, double y)
4  {
5    this->x = x;
6    this->y = y;
7  }
8
9  double Point::dist(Point p) const
10 {
11   return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
12 }
```

Inclusions multiples

- Parfois, des inclusions multiples de classes sont nécessaires
  - Un maillon de liste chaînée a un attribut maillon (élément suivant de la liste)
  - Un département est dirigé par un manager, un manager dirige un département

Un problème de taille

- Si un cycle d'attributs apparaît, les objets sont de taille infinie

- Solution : utiliser une adresse et `#ifndef` / `#define`

Exemple

- Fichier `mag-dep-pourri.cpp`

```
1  struct Manager
2  {
3    Departement& dpt;
4    string nom;
5
6    Manager(string nom, Departement& dpt) : nom(nom), dpt(dpt) {}
7  };
8
9  struct Departement
10 {
11   Manager mgr;
12   string nom;
13
14   Departement(string nom) : nom(nom) {}
15 };
16
17 int main()
18 {
19   Departement esi("ESI");
20   Manager mwi("Willemse", esi);
21   esi.mgr = mwi;
22 }
```

Exemple

- Fichiers `manager.*`, `departement.*` et `mag-dep-main.cpp`

```
1  #ifndef DEP
2  #define DEP
3
4  #include <string>
5
6  struct Manager;
7
8  struct Departement
9  {
10   Manager* mgr;//not allocated here
11   std::string nom;
12
13   Departement(std::string nom, Manager* mgr = nullptr);
14 };
15
16 #endif
```

```
1  #include "departement.h"
2
3  Departement::Departement(std::string nom, Manager* mgr) : nom(nom), mgr(mgr) {}
```

Exemple

- Fichiers `manager.*`, `departement.*` et `mag-dep-main.cpp`

```
1  #ifndef MAG
2  #define MAG
3
4  #include <string>
5  #include "departement.h"
6
7  struct Manager
8  {
9    Departement& dpt;
10   std::string nom;
11
12   Manager(std::string nom, Departement& dpt);
13 };
14
15 #endif
```

```
1  #include "manager.h"
2
3  Manager::Manager(std::string nom, Departement& dpt) : nom(nom), dpt(dpt) {}
```

Fonctions inline

- Fonction dont le corps est substitué à l'appel

Avantages

- Gain de temps (pas de `call`)

Inconvénients

- Exécutable grossit (copier / coller)

- Non contraignant : « demande courtoise »
- Ces fonctions n'ont *pas* d'adresse

Contraintes

- Une fonction `inline` est soit
  - déclarée avec le mot-clé `inline`
  - implémentée dans le prototype de la classe
  - une fonction `constexpr`

Remarque

- Doit être déclarée et implémentée au sein du même fichier
- Ne peut pas être utilisée là où l'adresse d'une fonction est attendue

Exemple

- Fichier `inline.h`

```
1  struct A
2  {
3    void f() //inline
4    {
5      cout << "Brol::f" << endl;
6    }
7  };
8
9  struct B
10 {
11   inline void f();//inline
12 };
13
14 void B::f()
15 {
16   cout << "Foo::f" << endl;//defined in same file
17 }
18
19 inline double sum(double a, double b) { return a + b; }
20
21 struct C
22 {
23   void f();//not inline
24 };
```

Exemple

- Fichiers `inline.cpp`, `inline-main.cpp`

```
1  void C::f()
2  {
3    cout << "C::f" << endl;
4  }
```

```
1  int main()
2  {
3    A a;
4    a.f();
5
6    B b;
7    b.f();
8
9    C c;
10   c.f();
11
12   cout << sum(2,3) << endl;
13 }
```

Membres statiques

Attribut statique

- Attribut de classe
- Déclaré avec le mot clé `static`
- L'initialisation se fait lors de la définition (en dehors de la déclaration de la classe)

Méthode statique

- Méthode de classe
- Déclaré avec le mot clé `static`
- Ne reçoit pas de point sur l'objet courant `this`
- Ne peut accéder qu'aux attributs statiques
- Elle travaille sur la classe (et non sur les objets)