

Ch. 7 - Fonctions amies

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique



6 octobre 2021

Table des matières

1 Introduction

2 Fonctions amies

3 Relations d'amitié

- Fonction indépendante amie d'une classe
- Fonction membre amie d'une classe
- Classe amie d'une classe

Table des matières

- 1 Introduction
- 2 Fonctions amies

- 3 Relations d'amitié
 - Fonction indépendante amie d'une classe
 - Fonction membre amie d'une classe
 - Classe amie d'une classe

Table des matières

1 Introduction

2 Fonctions amies

3 Relations d'amitié

- Fonction indépendante amie d'une classe
- Fonction membre amie d'une classe
- Classe amie d'une classe

Table des matières

- 1 Introduction
- 2 Fonctions amies
- 3 Relations d'amitié
 - Fonction indépendante amie d'une classe
 - Fonction membre amie d'une classe
 - Classe amie d'une classe

Table des matières

- 1 Introduction
- 2 Fonctions amies
- 3 Relations d'amitié
 - Fonction indépendante amie d'une classe
 - Fonction membre amie d'une classe
 - Classe amie d'une classe

Table des matières

- 1 Introduction
- 2 Fonctions amies
- 3 Relations d'amitié
 - Fonction indépendante amie d'une classe
 - Fonction membre amie d'une classe
 - Classe amie d'une classe

Introduction

Contexte

■ La POO « impose » l'encapsulation des données

- Les membres privés ne sont accessibles qu'à l'intérieur de la classe
- Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- Changer les membres en `public`
- Utiliser des accesseurs / mutateurs `inline`

Contexte

- La POO « impose » l'encapsulation des données
 - Les membres privés ne sont accessibles qu'à l'intérieur de la classe
 - Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- Changer les membres en `public`
- Utiliser des accesseurs / mutateurs `inline`

Contexte

- La POO « impose » l'encapsulation des données
 - Les membres privés ne sont accessibles qu'à l'intérieur de la classe
 - Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- Changer les membres en `public`
- Utiliser des accesseurs / mutateurs `inline`

Contexte

- La POO « impose » l'encapsulation des données
 - Les membres privés ne sont accessibles qu'à l'intérieur de la classe
 - Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- Changer les membres en `public`
- Utiliser des accesseurs / mutateurs `inline`

Contexte

- La POO « impose » l'encapsulation des données
 - Les membres privés ne sont accessibles qu'à l'intérieur de la classe
 - Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- Changer les membres en `public`
- Utiliser des accesseurs / mutateurs `inline`

Contexte

- La POO « impose » l'encapsulation des données
 - Les membres privés ne sont accessibles qu'à l'intérieur de la classe
 - Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- Changer les membres en `public`
- Utiliser des accesseurs / mutateurs `inline`

Contexte

- La POO « impose » l'encapsulation des données
 - Les membres privés ne sont accessibles qu'à l'intérieur de la classe
 - Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- 1 Changer les membres en `public`
- 2 Utiliser des accesseurs / mutateurs `inline`

Contexte

- La POO « impose » l'encapsulation des données
 - Les membres privés ne sont accessibles qu'à l'intérieur de la classe
 - Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- 1 Changer les membres en `public`
- 2 Utiliser des accesseurs / mutateurs `inline`

Contexte

- La POO « impose » l'encapsulation des données
 - Les membres privés ne sont accessibles qu'à l'intérieur de la classe
 - Usage de getters et setters

Inconvénient

- On peut être amené à faire beaucoup de `call`
- Légère perte de performance

Solutions connues

- 1 Changer les membres en `public`
- 2 Utiliser des accesseurs / mutateurs `inline`

Exemple de classe publique

■ Fichier public.cpp

```
1  class point
2  {
3      public :
4          double x,y;
5
6          point(double abs=0, double ord=0) : x(abs), y(ord) {}
7
8          double distance(point p)
9          {
10             return sqrt((x - p.x) * (x - p.x)
11                + (y - p.y) * (y - p.y));
12          }
13
14          string toString()
15          {
16             stringstream str;
17             str << "(" << x << ", " << y << ")";
18             return str.str();
19          }
20  };
```

Exemple de classe publique

■ Fichier `public.cpp`

```
1  class circle
2  {
3      point center;
4      double radius;
5
6      public :
7          circle(point p, double rad) : center(p), radius(rad) {}
8
9          inline void translate(double x, double y)
10         {
11             center.x += x;
12             center.y += y;
13         }
14
15         string toString()
16         {
17             stringstream str;
18             str << "Circle_of_center_" << center.toString() << "_and_of_radius_" << radius;
19             return str.str();
20         }
21     };
```

Exemple de classe publique

■ Fichier `public.cpp`

```
1  int main()
2  {
3      point p (1,1);
4      circle c (p, 2);
5
6      cout << p.toString() << endl;
7      cout << c.toString() << endl;
8
9      c.translate(1,-1);
10
11     cout << p.toString() << endl;
12     cout << c.toString() << endl;
13 }
```

Debriefing

Problèmes

- Violation d'encapsulation
- « Toute le monde » a accès en lecture et écriture sur les attributs

Avantages

- Gain de performances
- Écriture concise

Debriefing

Problèmes

- Violation d'encapsulation
- « Toute le monde » a accès en lecture et écriture sur les attributs

Avantages

- Gain de performances
- Écriture concise

Debriefing

Problèmes

- Violation d'encapsulation
- « Toute le monde » a accès en lecture et écriture sur les attributs

Avantages

- Gain de performances
- Écriture concise

Debriefing

Problèmes

- Violation d'encapsulation
- « Toute le monde » a accès en lecture et écriture sur les attributs

Avantages

- Gain de performances
 - Pas de `call`
- Écriture concise

Debriefing

Problèmes

- Violation d'encapsulation
- « Toute le monde » a accès en lecture et écriture sur les attributs

Avantages

- Gain de performances
 - Pas de `call`
- Écriture concise

Debriefing

Problèmes

- Violation d'encapsulation
- « Toute le monde » a accès en lecture et écriture sur les attributs

Avantages

- Gain de performances
 - Pas de `call`
- Écriture concise

Debriefing

Problèmes

- Violation d'encapsulation
- « Toute le monde » a accès en lecture et écriture sur les attributs

Avantages

- Gain de performances
 - Pas de `call`
- Écriture concise

Exemple de classe avec accesseurs / mutateurs

■ Fichier access.cpp

```
1  class point
2  {
3      double _x, _y; //ask why I wrote _x
4
5      public :
6          point(double abs=0, double ord=0) : _x(abs), _y(ord) {}
7
8          double distance(point p)
9          {
10             return sqrt((_x - p._x) * (_x - p._x)
11                + (_y - p._y) * (_y - p._y));
12          }
13
14          string toString()
15          {
16             stringstream str;
17             str << "(" << _x << ", " << _y << ")";
18             return str.str();
19          }
20
21          inline void set(int abs, int ord) { _x = abs; _y = ord; }
22          inline double x() const { return _x; }
23          inline double y() const { return _y; }
24  };
```

Exemple de classe avec accesseurs / mutateurs

■ Fichier access.cpp

```
1  class circle
2  {
3      point _center;
4      double _rad;
5
6      public :
7          circle(point p, double rad) : _center(p), _rad(rad) {}
8
9          void translate(double x, double y)
10         {
11             _center.set(_center.x() + x, _center.y() + y);
12         }
13
14         string toString()
15         {
16             stringstream str;
17             str << "Circle_of_center_" << _center.toString() << "_and_of_radius_" << _rad;
18             return str.str();
19         }
20     };
```

Debriefing

Problèmes

- Exécutable potentiellement plus gros

Avantages

- Encapsulation respectée
- Obligation de passer par des accesseurs / mutateurs
- Écriture assez concise

Debriefing

Problèmes

- Exécutable potentiellement plus gros

Avantages

- Encapsulation respectée
- Obligation de passer par des accesseurs / mutateurs
- Écriture assez concise

Debriefing

Problèmes

- Exécutable potentiellement plus gros

Avantages

- Encapsulation respectée
- Obligation de passer par des accesseurs / mutateurs
- Écriture assez concise

Debriefing

Problèmes

- Exécutable potentiellement plus gros

Avantages

- Encapsulation respectée
- Obligation de passer par des accesseurs / mutateurs
- Écriture assez concise

Debriefing

Problèmes

- Exécutable potentiellement plus gros

Avantages

- Encapsulation respectée
- Obligation de passer par des accesseurs / mutateurs
- Écriture assez concise

Debriefing

Problèmes

- Exécutable potentiellement plus gros

Avantages

- Encapsulation respectée
- Obligation de passer par des accesseurs / mutateurs
- Écriture assez concise

Fonctions amies

Overview

- Les solutions connues ont des inconvénients qui ne peuvent toujours être ignorés

Idée

- Possibilité de déclarer une fonction / classe *amie* à la définition d'une classe
- Autorise l'accès aux attributs privés pour cette fonction / classe
- Plusieurs situations d'amitié possibles
 - Fonction indépendante amie d'une classe
 - Fonction membre d'une classe A amie d'une classe B
 - Toutes les fonctions d'une classe sont amies d'une autre classe
- Utilisation du mot-clé `friend`

Overview

- Les solutions connues ont des inconvénients qui ne peuvent toujours être ignorés

Idée

- Possibilité de déclarer une fonction / classe *amie* à la définition d'une classe
- Autorise l'accès aux attributs privés pour cette fonction / classe
- Plusieurs situations d'amitié possibles
 - Fonction indépendante amie d'une classe
 - Fonction membre d'une classe A amie d'une classe B
 - Toutes les fonctions d'une classe sont amies d'une autre classe
- Utilisation du mot-clé `friend`

Overview

- Les solutions connues ont des inconvénients qui ne peuvent toujours être ignorés

Idée

- Possibilité de déclarer une fonction / classe *amie* à la définition d'une classe
- Autorise l'accès aux attributs privés pour cette fonction / classe
- Plusieurs situations d'amitié possibles
 - 1 Fonction indépendante amie d'une classe
 - 2 Fonction membre d'une classe A amie d'une classe B
 - 3 Toutes les fonctions d'une classe sont amies d'une autre classe
- Utilisation du mot-clé `friend`

Overview

- Les solutions connues ont des inconvénients qui ne peuvent toujours être ignorés

Idée

- Possibilité de déclarer une fonction / classe *amie* à la définition d'une classe
- Autorise l'accès aux attributs privés pour cette fonction / classe
- Plusieurs situations d'amitié possibles
 - 1 Fonction indépendante amie d'une classe
 - 2 Fonction membre d'une classe A amie d'une classe B
 - 3 Toutes les fonctions d'une classe sont amies d'une autre classe
- Utilisation du mot-clé `friend`

Overview

- Les solutions connues ont des inconvénients qui ne peuvent toujours être ignorés

Idée

- Possibilité de déclarer une fonction / classe *amie* à la définition d'une classe
- Autorise l'accès aux attributs privés pour cette fonction / classe
- Plusieurs situations d'amitié possibles
 - 1 Fonction indépendante amie d'une classe
 - 2 Fonction membre d'une classe A amie d'une classe B
 - 3 Toutes les fonctions d'une classe sont amies d'une autre classe
- Utilisation du mot-clé `friend`

Overview

- Les solutions connues ont des inconvénients qui ne peuvent toujours être ignorés

Idée

- Possibilité de déclarer une fonction / classe *amie* à la définition d'une classe
- Autorise l'accès aux attributs privés pour cette fonction / classe
- Plusieurs situations d'amitié possibles
 - 1 Fonction indépendante amie d'une classe
 - 2 Fonction membre d'une classe A amie d'une classe B
 - 3 Toutes les fonctions d'une classe sont amies d'une autre classe
- Utilisation du mot-clé `friend`

Overview

- Les solutions connues ont des inconvénients qui ne peuvent toujours être ignorés

Idée

- Possibilité de déclarer une fonction / classe *amie* à la définition d'une classe
- Autorise l'accès aux attributs privés pour cette fonction / classe
- Plusieurs situations d'amitié possibles
 - 1 Fonction indépendante amie d'une classe
 - 2 Fonction membre d'une classe A amie d'une classe B
 - 3 Toutes les fonctions d'une classe sont amies d'une autre classe
- Utilisation du mot-clé `friend`

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Avantage

- Le *concepteur* d'une classe spécifie quelles autres fonctions / classes sont ses amis
- Les amis ont accès aux membres privés comme s'ils étaient publics
- Très efficace
 - Pas de `call`
- Compromis d'encapsulation
 - Les attributs ne sont pas visibles depuis l'extérieur
 - ... sauf si le concepteur l'a explicitement autorisé

Remarque

- `friend` peut déclarer un prototype
- Attention à l'ordre et aux inclusions de fichier

Relations d'amitié

Remarques techniques

- Une déclaration d'amitié permet d'accéder aux membres privés (et protégés)
- Offre de meilleurs performances
- Une déclaration d'amitié peut déclarer un prototype de fonction
 - Mais pas un prototype de classe
- *Souvent*, l'endroit où une déclaration d'amitié au sein d'une classe est déclarée n'a pas d'importance
 - Avec des templates (cf. Ch. 13)

Hygiène de programmation

- Ne pas abuser

Remarques techniques

- Une déclaration d'amitié permet d'accéder aux membres privés (et protégés)
- Offre de meilleurs performances
- Une déclaration d'amitié peut déclarer un prototype de fonction
 - Mais pas un prototype de classe
- *Souvent*, l'endroit où une déclaration d'amitié au sein d'une classe est déclarée n'a pas d'importance
 - Avec des templates (cf. Ch. 13)

Hygiène de programmation

- Ne pas abuser

Remarques techniques

- Une déclaration d'amitié permet d'accéder aux membres privés (et protégés)
- Offre de meilleurs performances
- Une déclaration d'amitié peut déclarer un prototype de fonction
 - Mais pas un prototype de classe
- *Souvent*, l'endroit où une déclaration d'amitié au sein d'une classe est déclarée n'a pas d'importance
 - Avec des templates (cf. Ch. 13)

Hygiène de programmation

- Ne pas abuser

Remarques techniques

- Une déclaration d'amitié permet d'accéder aux membres privés (et protégés)
- Offre de meilleurs performances
- Une déclaration d'amitié peut déclarer un prototype de fonction
 - Mais pas un prototype de classe
- *Souvent*, l'endroit où une déclaration d'amitié au sein d'une classe est déclarée n'a pas d'importance
 - Avec des templates (cf. Ch. 13)

Hygiène de programmation

- Ne pas abuser

Remarques techniques

- Une déclaration d'amitié permet d'accéder aux membres privés (et protégés)
- Offre de meilleurs performances
- Une déclaration d'amitié peut déclarer un prototype de fonction
 - Mais pas un prototype de classe
- *Souvent*, l'endroit où une déclaration d'amitié au sein d'une classe est déclarée n'a pas d'importance
 - Avec des templates (cf. Ch. 13)

Hygiène de programmation

- Ne pas abuser

Remarques techniques

- Une déclaration d'amitié permet d'accéder aux membres privés (et protégés)
- Offre de meilleurs performances
- Une déclaration d'amitié peut déclarer un prototype de fonction
 - Mais pas un prototype de classe
- *Souvent*, l'endroit où une déclaration d'amitié au sein d'une classe est déclarée n'a pas d'importance
 - Avec des templates (cf. Ch. 13)

Hygiène de programmation

- Ne pas abuser

Remarques techniques

- Une déclaration d'amitié permet d'accéder aux membres privés (et protégés)
- Offre de meilleurs performances
- Une déclaration d'amitié peut déclarer un prototype de fonction
 - Mais pas un prototype de classe
- *Souvent*, l'endroit où une déclaration d'amitié au sein d'une classe est déclarée n'a pas d'importance
 - Avec des templates (cf. Ch. 13)

Hygiène de programmation

- Ne pas abuser

Remarques techniques

- Une déclaration d'amitié permet d'accéder aux membres privés (et protégés)
- Offre de meilleurs performances
- Une déclaration d'amitié peut déclarer un prototype de fonction
 - Mais pas un prototype de classe
- *Souvent*, l'endroit où une déclaration d'amitié au sein d'une classe est déclarée n'a pas d'importance
 - Avec des templates (cf. Ch. 13)

Hygiène de programmation

- Ne pas abuser

Table des matières

1 Introduction

2 Fonctions amies

3 Relations d'amitié

- **Fonction indépendante amie d'une classe**
- Fonction membre amie d'une classe
- Classe amie d'une classe

Exemple

■ Fichier indep.cpp

```
1  class point
2  {
3      int x, y;
4
5      public:
6          point(int abs = 0, int ord = 0) : x(abs), y(ord) {}
7
8          friend bool coincide(const point &, const point &);
9  };
10
11 int main()
12 {
13     point a(1,0), b(1), c;
14     if (coincide(a,b))
15         cout << "a_coincide_avec_b" << endl;
16     else
17         cout << "a_et_b_sont_différents" << endl;
18     if (coincide(a,c))
19         cout << "a_coincide_avec_c" << endl;
20     else
21         cout << "a_et_c_sont_différents" << endl;
22 }
23
24 bool coincide(const point & p, const point & q) { return ((p.x == q.x) && (p.y == q.y)); }
```


Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - 1. Savoir qu'une fonction indépendante est amie d'une classe
 - 2. Obtenir le prototype de la fonction indépendante
 - 3. Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - 1. Savoir qu'une fonction indépendante est amie d'une classe
 - 2. Obtenir le prototype de la fonction indépendante
 - 3. Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - Savoir qu'une fonction indépendante est amie d'une classe
 - Obtenir le prototype de la fonction indépendante
 - Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - Savoir qu'une fonction indépendante est amie d'une classe
 - Obtenir le prototype de la fonction indépendante
 - Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - Savoir qu'une fonction indépendante est amie d'une classe
 - Obtenir le prototype de la fonction indépendante
 - Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - Savoir qu'une fonction indépendante est amie d'une classe
 - Obtenir le prototype de la fonction indépendante
 - Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - Savoir qu'une fonction indépendante est amie d'une classe
 - Obtenir le prototype de la fonction indépendante
 - Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - 1 Savoir qu'une fonction indépendante est amie d'une classe
 - 2 Obtenir le prototype de la fonction indépendante
 - 3 Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - 1 Savoir qu'une fonction indépendante est amie d'une classe
 - 2 Obtenir le prototype de la fonction indépendante
 - 3 Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - 1 Savoir qu'une fonction indépendante est amie d'une classe
 - 2 Obtenir le prototype de la fonction indépendante
 - 3 Réécrire une fonction de même prototype

Remarques

- L'emplacement de la déclaration d'amitié de `coincide` au sein de `point` n'a pas d'importance
- Cette déclaration déclare également un prototype
 - Possibilité d'utiliser `coincide` dans `main` avant une « vraie » déclaration
- La déclaration d'amitié déclare une fonction indépendante car absence de paramètre implicite
 - Deux paramètres, pas de classe, etc.
- En général, une fonction amie d'une classe possède un ou plusieurs arguments du type de cette classe
 - Ce n'est pas une obligation
- Possibilité de viol d'encapsulation
 - 1 Savoir qu'une fonction indépendante est amie d'une classe
 - 2 Obtenir le prototype de la fonction indépendante
 - 3 Réécrire une fonction de même prototype

Table des matières

1 Introduction

2 Fonctions amies

3 Relations d'amitié

- Fonction indépendante amie d'une classe
- **Fonction membre amie d'une classe**
- Classe amie d'une classe

Exemple

■ Fichier `member.cpp`

```
1  class A; // forward declaration of A needed by B
2
3  class B
4  {
5      public:
6          void fB(A& a);
7  };
8
9  class A
10 {
11     int i;
12     public:
13         // specifying function fA as a friend of A, fA is not member function of A
14         friend void fA(A& a);
15
16         // specifying B class member function fB as a friend of A
17         friend void B::fB(A& a);
18     };

```

Exemple

■ Fichier member.cpp

```
1 // fA is Friend function of A
2 void fA(A& a)
3 {
4     a.i = 11; // accessing and modifying Class A private member i
5     cout << a.i << endl;
6 }
7
8 // B::fB should be defined after class A definition
9 void B::fB(A& a)
10 {
11     a.i = 22;
12     cout << a.i << endl;
13 }
14
15 int main()
16 {
17     A a;
18     fA(a); // calling friend function of class A
19
20     B b;
21     b.fB(a); // calling B class member function fB, B::fB is friend of class A
22 }
```

Table des matières

1 Introduction

2 Fonctions amies

3 Relations d'amitié

- Fonction indépendante amie d'une classe
- Fonction membre amie d'une classe
- Classe amie d'une classe

Exemple

■ Fichier class.cpp

```
1  class B;
2
3  class A
4  {
5      int _i;
6
7      public:
8          A(int i) : _i(i) {}
9          int i() const { return _i; }
10
11         friend class B;
12 };
13
14 class B
15 {
16     A a;
17     int _j;
18
19     public:
20         B(A a, int j) : a(a), _j(j) {}
21
22         int broI() const { return a._i * _j; }
23 };
```


Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
- L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
- À partir de C++11, les amis ont accès aux classes internes privées
- Souvent, un choix de design est effectué pour soit
 - rendre une classe B entière amie d'une autre classe A
 - faire d'une classe B une classe interne d'une autre classe A
- Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
- L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
- À partir de C++11, les amis ont accès aux classes internes privées
- Souvent, un choix de design est effectué pour soit
 - rendre une classe B entière amie d'une autre classe A
 - faire d'une classe B une classe interne d'une autre classe A
- Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
- L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
- À partir de C++11, les amis ont accès aux classes internes privées
- Souvent, un choix de design est effectué pour soit
 - rendre une classe B entière amie d'une autre classe A
 - faire d'une classe B une classe interne d'une autre classe A
- Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
- L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
- À partir de C++11, les amis ont accès aux classes internes privées
- Souvent, un choix de design est effectué pour soit
 - rendre une classe B entière amie d'une autre classe A
 - faire d'une classe B une classe interne d'une autre classe A
- Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
- L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
- À partir de C++11, les amis ont accès aux classes internes privées
- Souvent, un choix de design est effectué pour soit
 - rendre une classe B entière amie d'une autre classe A
 - faire d'une classe B une classe interne d'une autre classe A
- Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
- L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
- À partir de C++11, les amis ont accès aux classes internes privées
- Souvent, un choix de design est effectué pour soit
 - rendre une classe B entière amie d'une autre classe A
 - faire d'une classe B une classe interne d'une autre classe A
- Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
- L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
- À partir de C++11, les amis ont accès aux classes internes privées
- Souvent, un choix de design est effectué pour soit
 - rendre une classe B entière amie d'une autre classe A
 - faire d'une classe B une classe interne d'une autre classe A
- Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
- L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
- À partir de C++11, les amis ont accès aux classes internes privées
- Souvent, un choix de design est effectué pour soit
 - 1 rendre une classe B entière amie d'une autre classe A
 - 2 faire d'une classe B une classe interne d'une autre classe A
- Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
 - L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
 - À partir de C++11, les amis ont accès aux classes internes privées
-
- Souvent, un choix de design est effectué pour soit
 - 1 rendre une classe B entière amie d'une autre classe A
 - 2 faire d'une classe B une classe interne d'une autre classe A
 - Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
 - L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
 - À partir de C++11, les amis ont accès aux classes internes privées
-
- Souvent, un choix de design est effectué pour soit
 - 1 rendre une classe `B` entière amie d'une autre classe `A`
 - 2 faire d'une classe `B` une classe interne d'une autre classe `A`
 - Règles particulières dans le cas de templates (Cf. Ch. 13)

Propagation d'une relation d'amitié

Règles

- Les relations d'amitié ne sont pas transitives
 - L'ami d'un ami n'est pas votre ami
 - L'amitié n'est pas propagée par héritage (Cf. Ch. 11)
 - Les enfants de votre ami ne sont pas vos amis
 - Vos enfants ne sont pas les amis de votre ami
 - À partir de C++11, les amis ont accès aux classes internes privées
-
- Souvent, un choix de design est effectué pour soit
 - 1 rendre une classe `B` entière amie d'une autre classe `A`
 - 2 faire d'une classe `B` une classe interne d'une autre classe `A`
 - Règles particulières dans le cas de templates (Cf. Ch. 13)

Exemple

■ Fichier subclassing.cpp

```
1  class A
2  {
3      int _i;
4
5      public:
6          A() : _i(2) {}
7          int i() const { return _i; }
8
9          friend class M; //class M is a friend of A
10 };
11
12 class B : public A //A is a subclass of A
13 {
14     int _j;
15
16     public:
17         B() : _j(3) {}
18         int j() const { return _j; }
19 };
```

Exemple

■ Fichier subclassing.cpp

```
1 //M is a friend of A and not a friend of its children
2 class M
3 {
4     int _k;
5
6     public:
7         M(A a) : _k(a._i * 2) {}
8         //M(B b) : _k(b._j * 3) {}
9
10        int k() const { return _k; }
11};
12
13 //children of M are neither friends of A or B
14 class N : public M
15 {
16     int _l;
17
18     public:
19         N(A a) : M(a) /*, _l(a._i * 4)*/ {}
20         N(B b) : M(b) /*, _l(b._j * 5)*/ {}
21
22        int l() const { return _l; }
23};
```