Chapitre 7

C++ avancé

Héritage multiple

Espaces de noms

Fonctions anonymes

Classes emboîtées

Modèle de classes

Héritage multiple

Héritage multiple

Une classe peut avoir plusieurs classes-mères.

Syntaxe Héritage multiple

```
class clfille: public clmere1, public clmere2 ... {
...
};
```

Problème

Que se passe-t-il si les classes-mères définissent des attributs de même nom (éventuellement de types différents) ou des méthodes de même nom et de même signature ?

⇒ L'héritage multiple peut conduire à la définition de classes dont le comportement est peu intuitif.
⇒ Il est conseillé (à moins de très bien connaître C++) d'utiliser l'héritage multiple uniquement dans des cas où il n'y a pas de problèmes.

Héritage multiple - Quand l'utiliser ?

Principalement pour implémenter des interfaces.

- Une interface est une classe abstraite définissant uniquement des méthodes virtuelles pures (et un destructeur virtuel).
- Une classe peut implémenter plusieurs interfaces et être, en plus, une sous-classe d'une classe « classique ».
- En C++, on utilise l'héritage multiple pour représenter le fait que la classe implémente des interfaces.

Héritage multiple - Exemple

Déclaration d'une interface (heritagemultiple/pile.hh)

```
class debug {
  public:
  virtual ~debug( ) =default;
  virtual unsigned int tailleoctets( ) const =0;
};
```

Héritage multiple - Exemple

Déclaration d'une interface (heritagemultiple/pile.hh)

```
class pile {
  public:
  virtual ~pile() =default;
  virtual void empiler(int i) =0;
  virtual bool vide() const =0;
  virtual void depiler() =0;
  virtual int sommet() const =0;
};
```

Héritage multiple - Exemple

Classe implémentant 2 interfaces

(heritagemultiple/pile.hh)

```
class pilevector: public debug, public pile {
  public:
  unsigned int tailleoctets() const override;
  void empiler(int i) override;
  bool vide() const override;
  void depiler() override;
  int sommet() const override;
  public:
  std::vector<int> _contenu;
};
```

Héritage multiple - Exemple

Définition de la classe

(heritagemultiple/pile.cc)

```
unsigned int pilevector::tailleoctets() const {
  return _contenu.size() * sizeof(int);
}
void pilevector::empiler(int i) {
  _contenu.push_back(i);
}
bool pilevector::vide() const {
  return _contenu.empty();
}
void pilevector::depiler() {
  _contenu.pop_back();
}
int pilevector::sommet() const {
  return _contenu.back();
}
```

Héritage multiple - Exemple

- `pilevector` implémente les deux interfaces `debug` et `pile`.
- Toute instance de `pilevector` peut être passée comme paramètre à une fonction/méthode qui attend un `debug` ou un `pile`.

Utilisation d'une interface

(heritagemultiple/pile.cc)

```
void remplirpile(pile & p) {
  for (int i=0; i<10; ++i)
    p.empiler(i);
}
void viderpile(pile & p) {
  while (!p.vide())
    p.depiler();
}
void affichertaille(debug const & o) {
  std::cout << "Taille de l'objet " << o.tailleoctets() << "\n";
}
```

Héritage multiple - Un cas à problème

`pilevector` est un cas sans problèmes : les super-classes n'ont pas de membres ayant le même nom.

Un cas à problème

(heritagemultiple/probleme.cc)

```
class A {
  public:
  int attribut;
  void meth() {}
};

class B {
  public:
  float attribut;
  int meth() {return 0;}
};

class C: public A, public B {
};
```

Héritage multiple - Un cas à problème

Exemple

```
C c;
c.attribut = 0;
c.meth();
```

Erreur de compilation :

```
request for member 'attribut' is ambiguous
candidates are: float B::attribut    int A::attribut
request for member 'meth' is ambiguous
candidates are: int B::meth()        void A::meth()
```

Attention

La classe `C` a hérité de deux attributs nommés `attribut` et de deux méthodes nommées `meth`.

Héritage multiple

Il faut préciser quel membre doit être utilisé...

En préfixant son nom par le nom de la classe dont il est issu.

Exemple

```
c.A::attribut = 0;
c.B::meth();
```

Cela peut conduire à du code difficile à mettre au point.

Espaces de noms

Espaces de noms

- Les espaces de noms permettent de « ranger » des classes dans des ensembles.
- Chaque ensemble a un nom, et deux ensembles différents peuvent contenir une classe de même nom.
  Le nom complet d'une classe est en fait `nomespace::nomclasse`.
- Par exemple, l'espace de nom `std` peut contenir une classe `string`, et l'espace de nom `mabibliotheque` une classe `string`.
- Les espaces de noms sont habituellement utilisés lors du développement d'un code prévu pour être réutilisable, et importé dans d'autres projets (bibliothèque de classes)
  Si le projet qui importe le code contient une classe de même nom, erreur de compilation.
  Avec l'utilisation d'espaces de noms, pas d'ambiguïté.

Espaces de noms

Syntaxe Espaces de noms

- Déclaration d'une classe (dans un `.hh`)

```
namespace nomespace {
  class nomclasse { ... };
}
```

- Définition d'une classe (dans un `.cc`)

```
namespace nomespace {
  nomclasse::nomclasse() { ... }
}
```

Espaces de noms

Syntaxe Espaces de noms

- Utilisation d'une classe

  `nomespace::nomclasse`
- Recherche des classes dans un espace de noms (import)

  `using namespace nomespace;`

  À éviter dans un fichier `.hh`
- Import d'un élément d'un espace

  `using nomespace::nomelement;`

  (exemple : `using std::cout;`)

Fonctions anonymes

Fonction anonyme

Une fonction anonyme est une fonction… qui n'a pas de nom.

- Ne peut être appelée ailleurs qu'à l'endroit où elle a été définie.
- Fonction courte à usage unique.
- Peut être passée comme paramètre à une fonction/méthode.
- Peut accéder à des variables de la fonction/méthode dans laquelle elle est déclarée.

Fonction anonyme

Syntaxe Fonction anonyme

- `[captures](parametres)->typederetour{corps}`
- `[captures](parametres){corps}`

Si le type de retour n'est pas présent, il est déterminé automatiquement par le compilateur à partir des types des valeurs retournées par les **`return`** de la fonction (qui doivent être de même type...idéalement un seul **`return`**).

Fonction anonyme - Exemple

La bibliothèque standard C++ fournit un certain nombre de services qui prennent comme paramètre une fonction qui sera exécutée sur les éléments d'un conteneur.

Certains de ces services sont des méthodes du conteneur (`std::list::remove_if`, `std::list::sort`...) d'autres sont des fonctions pour la plupart disponibles dans la bibliothèque d'algorithmes `http://en.cppreference.com/w/cpp/algorithm`.

Par exemple `std::for_each` exécute une fonction sur tous les éléments d'un conteneur compris dans un intervalle défini par couple d'itérateurs.

Fonction anonyme - Exemple

Soit v un `std::vector<int>`.

Fonction anonyme - Déclaration complète (lambdas/lambdas.cc)

```
std::for_each(v.begin(), v.end(),
  [](int i)->void{std::cout << i << ' ';});
```

Fonction anonyme - Simplification (lambdas/lambdas.cc)

```
std::for_each(v.begin(), v.end(),
  [](int i){std::cout << i << ' ';});
```

Fonction anonyme - Simplification (lambdas/lambdas.cc)

```
std::for_each(v.begin(), v.end(),
  [](auto i){std::cout << i << ' ';});
```

Fonction anonyme - Exemple

Le paramètre peut aussi être passé par référence.

Fonction anonyme

(lambdas/lambdas.cc)

```
void doubler(std::vector<int> & v) {
  std::for_each(v.begin(), v.end(), [](auto & i){i*=2;});
}
```

Fonction anonyme - Exemple

Utilisation du type de retour de la fonction

`std::all_of` retourne un booléen valant **`true`** si tous les éléments repérés par un couple d'itérateurs vérifient le prédicat passé en troisième paramètre (la fonction prédicat retourne **`true`**), **`false`** sinon.

Fonction anonyme

(lambdas/lambdas.cc)

```
bool touspositifs(std::vector<int> const & v) {
  return std::all_of(v.begin(), v.end(),
    [](auto i){ return i >= 0; });
}
```

Il y a aussi `std::any_of` et `std::none_of`.

Fonction anonyme - Capture

La capture permet de fournir à une fonction anonyme des données supplémentaires, quand la signature est fixée.

Exemple. Utiliser `std::any_of` pour déterminer si un `vector<int>` contient (au moins) une valeur supérieure à une valeur `borne`.Utiliser un paramètre de la fonction ? **Non** . La signature du prédicat est fixée par `std::any_of` : ce doit être `(int)->bool`.

Fonction anonyme - Capture

La capture est composée d'une liste de variables accessibles dans le bloc dans laquelle la fonction anonyme est déclarée, séparées par des virgules, passées par valeur ou par référence, dans ce cas, elles sont préfixées par &.

Les variables de la capture sont accessibles dans la fonction anonyme.

Fonction anonyme - Capture

(lambdas/lambdas.cc)

```
bool contientsuperieur(std::vector<int> const & v, int borne) {
  return std::any_of(v.begin(), v.end(),
    [borne](auto i){ return i >= borne; });
}
```

Fonction anonyme - Capture (lambdas/lambdas.cc)

```
int somme(std::vector<int> const & v) {
  int result(0);
  std::for_each(v.begin(), v.end(), [&result](auto i){result+=i;});
  return result;
}
```

On ne peut pas capturer un attribut d'une classe, mais on peut capturer **this**, qui donne accès aux attributs privés.

Classes emboîtées

Classes emboîtées

- Il est possible de définir des classes (ou types : **`struct`**, **`union`**, **`enum`**) à l'intérieur d'une classe.
- La visibilité (encapsulation) s'applique à ces types.
- Cela est utilisé pour définir une classe qui n'a de sens qu'« à l'intérieur » d'une autre classe.

Exemple. Un graphe (orienté) est composé de sommets, chaque sommet contient un ensemble d'arcs sortants.
Les sommets n'ont de sens qu'à l'intérieur d'un graphe. Les arcs sortants n'ont de sens qu'à l'intérieur d'un sommet.

Classes emboîtées

Déclarations emboîtées

(emboites/graphe.hh)

```
class graphe {
  public:
  class sommet {
    public:
    using identifiant = unsigned int;
    private:
    struct arcsortant {
      identifiant _extremite;
      std::string _etiquette;
    };
    public:
      sommet(std::string const & et)
        :_etiquette(et), _id(++_compteur) {}
      identifiant id() const { return _id; }
      std::vector<identifiant> voisins() const;
      std::string const & etiquette() const { return _etiquette; }
      void ajouterarc(std::string const & et,identifiant extremite);
```

Déclarations emboîtées (suite `sommet`)

(emboites/graphe.hh)

```
    private:
      std::string _etiquette;
      identifiant _id;
      std::list<arcsortant> _arcs;
      static identifiant _compteur;
  };
  public:
    sommet::identifiant ajoutersommet(std::string const & et);
    sommet const & accessommet(sommet::identifiant id) const;
    void ajouterarc(sommet::identifiant origine, sommet::identifiant
         extremite, std::string const & etiquette);
  private:
    std::list<sommet>::iterator chercher(sommet::identifiant id);
    std::list<sommet>::const_iterator chercher(sommet::identifiant
        id) const;
  private:
    std::list<sommet> _sommets;
};
```

Classes emboîtées

Définition des classes (emboites/graphe.cc)

```
graphe::sommet::identifiant graphe::sommet::_compteur(0);

void graphe::sommet::ajouterarc(std::string const & et, identifiant
    extremite) {
  auto f(std::find_if(_arcs.begin(), _arcs.end(), [extremite](auto
      const & a){return a._extremite==extremite;}));
  if (f == _arcs.end())
    _arcs.push_back(arcsortant {extremite, et});
  else
    *f = arcsortant {extremite, et};
}

std::vector<graphe::sommet::identifiant> graphe::sommet::voisins()
    const {
  std::vector<identifiant> result;
  for (auto const & av : _arcs)
    result.push_back(av._extremite);
  return result;
}
```

Classes emboîtées

Définition des classes

(emboites/graphe.cc)

```
graphe::sommet::identifiant graphe::ajoutersommet(std::string const
    & et) {
  sommet nouv(et);
  _sommets.push_back(nouv);
  return nouv.id();
}

graphe::sommet const & graphe::accessommet(sommet::identifiant id)
    const {
  return *chercher(id);
}

void graphe::ajouterarc(sommet::identifiant origine, sommet::
    identifiant extremite, std::string const & etiquette) {
  auto io(chercher(origine));
  auto ie(chercher(extremite));
  io->ajouterarc(etiquette, extremite);
}
```

Classes emboîtées

Définition des classes

(emboites/graphe.cc)

```
std::list<graphe::sommet>::iterator graphe::chercher(sommet::
    identifiant id) {
  auto f = std::find_if(_sommets.begin(), _sommets.end(), [id](auto
      const & a){return a.id() == id;});
  if (f == _sommets.end())
    throw std::invalid_argument(std::to_string(id));
  else
    return f;
}

std::list<graphe::sommet>::const_iterator graphe::chercher(sommet::
    identifiant id) const {
  auto f = std::find_if(_sommets.begin(), _sommets.end(), [id](auto
      const & a){return a.id() == id;});
  if (f == _sommets.end())
    throw std::invalid_argument(std::to_string(id));
  else
    return f;
}
```

Classes emboîtées

Exemple d'utilisation (emboites/graphe.cc)

```
int main() {
  graphe g;
  graphe::sommet::identifiant paris(g.ajoutersommet("Paris"));
  auto angers(g.ajoutersommet("Angers"));
  auto nantes(g.ajoutersommet("Nantes"));
  g.ajouterarc(angers, nantes, "Bus");
  g.ajouterarc(nantes, angers, "Bus");
  g.ajouterarc(angers, paris, "Train");
  g.ajouterarc(paris, nantes, "Avion");
  std::cout << "Voisins de " << g.accessommet(angers).etiquette() <<
       "\n";
  auto voisinsangers(g.accessommet(angers).voisins());
  for (auto v : voisinsangers)
    std::cout << g.accessommet(v).etiquette() << " ";
  return 0;
}
```

Modèle de classes

Modèle de classes

- Un modèle de classes (ou patron de classes) permet de construire plusieurs classes qui diffèrent par l'utilisation d'un type de données.
- `std::vector` est un modèle qui permet de définir les classes `std::vector<int>`, `std::vector<std::string>`, ...

Attention

Un modèle de classes n'est pas une classe.

Il ne peut donc pas être instancié ou utilisé comme un type.

Exemple (Erroné)

```
std::vector v;
v.push_back(???);
```

Les conteneurs de la STL sont des modèles de classes. Il est possible de définir ses propres modèles.

Modèle de classes

Déclaration modèle de classes (template/pile.hh)

```
#pragma once
#include <vector>

template <typename t>
class pile {
  public:
  bool vide( ) const {
    return _contenu.empty( ); }
  void empiler(t v) {
    _contenu.push_back(v); }
  void depiler( ) {
    _contenu.pop_back( ); }
  t sommet( ) const;
  private:
  std::vector<t> _contenu;
};
```

Modèle de classes

Déclaration modèle de classes (template/pile.hh)

```
template <typename t>
t pile<t>::sommet() const {
  t s = _contenu.back();
  return s;
}
```

Modèle de classes

Utilisation modèle de classes (template/pilemain.cc)

```
#include "pile.hh"
#include <string>

int main() {
  pile<int> pint; pile<std::string> pstring;
  pint.empiler(2);
  pstring.empiler("essai");
  return 0;
}
```

Modèle de classes

- Les modèles de classes sont souvent utilisés pour définir des classes conteneurs.
- Un modèle peut avoir plusieurs arguments.
- La définition des méthodes est habituellement donnée dans le fichier `.hh`
  - dans la déclaration de la classe.
  - après la déclaration de la classe.
  - dans un fichier d'implémentation du modèle, inclus dans le fichier `.hh`.