Programmation C++ Avancée

Session 2 – Objets : Modèle et Cycle de vie

Joel Falcou Guillaume Melquiond

Laboratoire de Recherche en Informatique

Service, Interface, Contrat ?

Un objet – vision logique

- Un objet encapsule un état
- Un objet propose un service
- Un objet satisfait à une interface

Un objet – vision physique

- un état = données membres
- un comportement = fonctions membres
- le tout défini dans une `class`

Syntaxe de base

```
class simple_class
{
  std::vector<float>  buffer;

  protected:
  int                 id;
  std::string         name;

  public:
  simple_class();
  simple_class(simple_class const &other);
  ~simple_class();

  simple_class &operator=(simple_class const &other);
  int update(double n);
};
```

Héritage public

Définition

- Une classe hérite d'une autre afin de spécialiser son comportement
- La classe "fille" accède aux données et à l'interface publique de sa "mère"
- Notion de sous-classe

```
class base
{
  public:
  virtual void behavior();
};
```

Héritage public

Définition

- Une classe hérite d'une autre afin de spécialiser son comportement
- La classe "fille" accède aux données et à l'interface publique de sa "mère"
- Notion de sous-classe

```
class derived : public base
{
  public:
  virtual void behavior();
  void derived_behavior();
};
```

Héritage public

Définition

- Une classe hérite d'une autre afin de spécialiser son comportement
- La classe "fille" accède aux données et à l'interface publique de sa "mère"
- Notion de sous-classe

```
void process(base &b)
{
  b.behavior();
}
```

Héritage public

Définition

- Une classe hérite d'une autre afin de spécialiser son comportement
- La classe "fille" accède aux données et à l'interface publique de sa "mère"
- Notion de sous-classe

```
int main()
{
  derived d;

  d.behavior();
  d.derived_behavior();

  process(d);
}
```

Gestion du polymorphisme

```
class base
{
  public:
  virtual void behavior();
};

class derived : public base
{
  public:
  virtual void behavior();
  void derived_behavior();
};
```

Gestion du polymorphisme

```
class base
{
  public:
  virtual void behavior();
  virtual void foo() final {}
};

class derived final : public base
{
  public:
  virtual void behavior() override;
  void derived_behavior() {}
};
```

Principe de substitution de Liskov

Énoncé

Partout où un objet x de type T est attendu, on doit pouvoir passer un objet y de type U, avec U héritant de T.

Traduction

- une classe = une interface = un **contrat**
- Les pré-conditions ne peuvent être qu'affaiblies
- Les post-conditions ne peuvent être que renforcées

Principe de substitution de Liskov

```
class rectangle
{
  protected:
  double width;
  double height;

  public:
  rectangle() : width(0), height(0) {}
  virtual void set_width(double x) { width = x; }
  virtual void set_height(double x) { height = x; }
  double area() const { return width * height; }
};
```

Principe de substitution de Liskov

```
class square : public rectangle
{
  public:
  void set_width(double x)
  {
    rectangle::set_width(x);
    rectangle::set_height(x);
  }

  void set_height(double x)
  {
    rectangle::set_width(x);
    rectangle::set_height(x);
 }
};
```

Principe de substitution de Liskov

```
void foo(rectangle &r)
{
  r.set_height(4);
  r.set_width(5);

  if (r.area() != 20)
    std::cout << "ERROR " << r.area() << " != 20\n";
}

int main()
{
  rectangle r;
  square s;

  foo(r);
  foo(s);
}
```

Héritage privé

Définition

- Résout le problème de la factorisation de code
- Permet la réutilisation des composants logiciels
- Pas de relation de sous-classe

Héritage privé

```
class stack : private std::vector<double>
{
  public:
  using parent = std::vector<double>;
  using parent::size;

  void push(double v) { parent::push_back(v); }
  double top() { return parent::back(); }

  double pop()
  {
    double v = parent::back();
    parent::pop_back();
    return v;
  }
};
```

Où ranger les objets ?

La pile

- Mémoire rapide mais limitée en espace et en temps
- Nettoyage automatique en fin de bloc
- Simple et efficace

Le tas

- Espace virtuellement illimité
- Accessible via **`new`** et **`delete`**
- Nécessite une gestion fine (voir Session 3)

Initialisation d'un objet

Constructeur

- Chargé d'initialiser toutes les données membres d'un objet alloué
- Initialisation d'abord des données membres des classes "mères" via l'appel des constructeurs de ces classes
- Initialisation ensuite des données membres de la classe courante via l'appel des constructeurs des types de ces membres
- Enfin exécution d'un bloc de code

Initialisation d'un objet

```
struct C : A, B {
  T1 d1;
  T2 d2;
  C(int x, T2 const &y) :
    // initialisation des classes meres
    A(x), B(),
    // initialisation des donnees membres
    d1(x + y, "abc"), d2(y)
  {
    // code arbitraire
    d1.f(x);
  }
};

int main() {
  C z(5, T2());
}
```

Gestion des temporaires

lvalue vs *rvalue*

- lvalue : objet avec une identité, un nom
- rvalue : objet sans identité
- La durée de vie d'une rvalue est en général bornée au `statement`
- Une rvalue peut survivre dans une référence vers une lvalue constante

```
int a = 42;       // lvalue
int b = 43;       // lvalue
a = b;            // ok
a = a * b;        // ok
a * b = 42;       // erreur

int getx() { return 17;}
int &lr  = getx();          // erreur
```

Référence vers rvalue

Objectifs

- Discriminer via un qualificateur lvalue et rvalue
- Expliciter les opportunités d'optimisation
- Simplifier la définition d'interfaces

Notation

- `T&` : référence vers lvalue
- `T const&` : référence vers lvalue constante
- `T&&` : référence vers rvalue

Référence vers rvalue

Exemple

```
void foo(int const &) { std::cout << "lvalue\n"; }
void foo(int &&x)     { std::cout << "rvalue\n"; }
int bar()             { return 1337; }

int main()
{
  int  x = 3;
  int &y = x;

  foo(x);
  foo(y);
  foo(4);
  foo(bar());
}
```

Référence vers rvalue

Le problème du forwarding

```
void foo(int const &) { std::cout << "lvalue\n"; }
void foo(int &&)      { std::cout << "rvalue\n"; }
void chu(int &&x)     { foo(x); }
int bar()             { return 1337; }

foo(bar());
chu(bar());
```

Référence vers rvalue

Le problème du forwarding

```
void foo(int const &) { std::cout << "lvalue\n"; }
void foo(int &&)      { std::cout << "rvalue\n"; }
void chu(int &&x)     { foo(std::forward<int>(x)); }
int bar()             { return 1337; }

foo(bar());
chu(bar());
```

Sémantique de transfert

Problématique

- Copier un objet contenant des ressources est coûteux
- Copier depuis un temporaire est doublement coûteux (allocation + désallocation)
- Limite l'expressivité de certaines interfaces
- Pourquoi ne pas recycler le temporaire ?

Solution

- Utiliser les rvalue-references pour détecter un temporaire
- Extraire son contenu et le transférer dans un objet pérenne
- Stratégie généralisée à tout le langage et à la bibliothèque standard

Sémantique de transfert

```
std::vector<int> sort(std::vector<int> const &v)
{
  std::vector<int> that{v};

  std::sort(that.begin(), that.end());

  return that;
}
```

Sémantique de transfert

```
void sort(std::vector<int> const &v, std::vector<int> &that)
{
  that = v;
  std::sort(that.begin(), that.end());
}
```

Sémantique de transfert

```
std::vector<int> sort(std::vector<int> &&v)
{
  std::vector<int> that{std::move(v)};

  std::sort(that.begin(), that.end());

  return that;
}
```

Sémantique de transfert

```
std::vector<int> sort(std::vector<int> v)
{
  std::sort(v.begin(), v.end());
  return v;
}
```