C++ avancé

Jacques Farré

Jacques.Farre@unice.fr

deptinfo.unice.fr/~jf/C++/AV

Modalités

Cours sur 6 semaines :
- vendredi 14h-17h45
- 1,5h de cours, 2h TP

Contrôles :
- **sans documents ni device électronique**
  - questions sur le cours et les TP
- 1/2h début mai
- 1h fin mai

Plan du cours

- Quelques rappels de base et quelques points pas encore vus
- Conception de classes robustes et programmation dirigée par les types
- Les principales innovations de C++ 2011
- Étude de la conception de quelques classes de la bibliothèque BOOST

Quelques rappels de base et points nouveaux

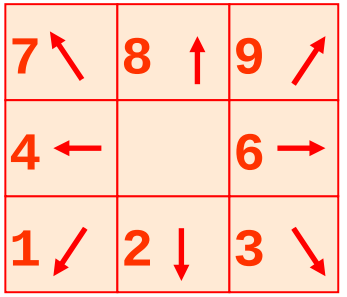
- Pointeurs sur membre
- Rappels sur les classes C++
- Rappels sur la construction
- Héritage multiple

Pointeurs sur membre (utilité)

Soit un objet graphique

```
class ObjetGraphique {
public:
 ObjetGraphique(int xx,int yy) : x(xx), y(yy) { }
 void Haut() {--y;}
 void Bas(){++y;}
 void Gauche() {--x;}
 void Droite(){++x;}
 ...
protected:
 int x; int y; // coordonnées
};
```

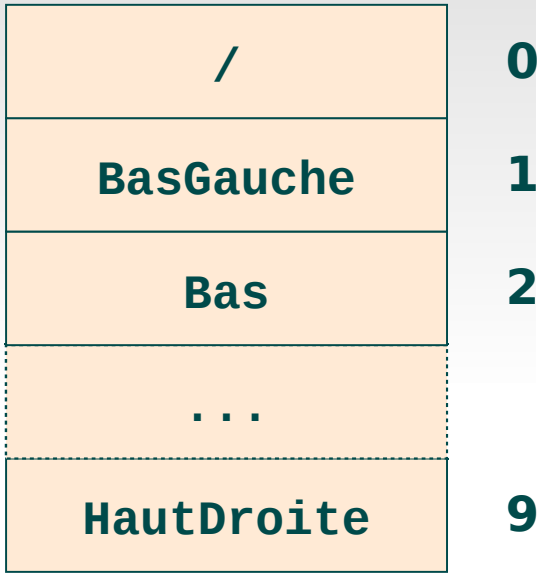
Lier les codes du clavier aux déplacements de l'objet



Pointeurs sur membre (principes)

Idée: utiliser un tableau de mouvements indexé par les touches du clavier :

- on peut définir des fonctions que l'on mettra dans un tableau de pointeurs sur fonctions




```
void MBasGauche(ObjetGraphique* o) {
  o->BasGauche();
}
```

Pointeurs sur membre (détour par les fonctions)

On peut définir, par exemple

```
typedef void (*Pmvt)(ObjetGraphique*);
```

Pmvt est un type pointeur sur des fonctions de signature `void F (ObjetGraphique*);`

- le tableau se déclare comme suit

```
static Pmvt mvt[]=
        {0,&MBasGauche,…,&MHautDroite};
```

- et s'utilise ainsi sur un objet ob :

```
(*mvt[touche]) (ob);
```

Pointeurs sur méthode (déclaration)

On a simplement encapsulé les appels de méthodes dans des (pointeurs sur) fonctions

Pourquoi ne pas le faire directement ?

- on veut des pointeurs sur des méthodes d'`ObjetGraphique`, qui ont pour signature `void M();`
- donc un typedef pour la lisibilité :
  `typedef void (ObjetGraphique::*Pmvt)();`

Pointeurs sur méthode (utilisation)

Tableau des mouvements :

```
typedef void (ObjetGraphique::*Pmvt)();
  static Pmvt mvt[] = {
    0,
&ObjetGraphique::BasGauche,
…,
&ObjetGraphique:: HautDroite
};
```

- Utilisation sur un objet o ou un pointeur sur un objet p :

  ```
  Pmvt m = mvt[code]; (o.*m)(); (p->*m)();
  ```

- ou plus directement : `(o.*mvt[code])();`

Pointeurs sur méthode (et liaison dynamique)

Les pointeurs sur méthode sont compatibles avec la liaison dynamique:

```
class ObjetGraphique {
public:
 …
 virtual void Haut() {--y;}
};

class ObjetGraphiquePrisonnier : public ObjetGraphique {
public:
 …
 void Haut() { if (y > mur_du_haut) --y; }
};
```

```
void (ObjetGraphique::*p)()= &ObjetGraphique::Haut;
…
ObjetGraphique* g = new ObjetGraphiquePrisonnier(…);
…
(g->*p)(); // ObjetGraphiquePrisonnier::Haut
```

Pointeurs sur membre (attributs)

- Exemple de déclaration:

```
int A::*p;    // pointeur sur un
              // attribut entier de A
```

- Initialisation:

```
p = &A::x; // p pointe les x de A
```

- Mettre à 0 les attributs **x** d'un tableau **t** de **A** : `for (i=0; i<N; ++i) t[i].*p = 0;`
- Mettre à 0 l'attribut **x** du **A** pointé par **q** :

```
q->*p = 0;
```

Les multiples facettes d'une classe C++

- Un modèle pour créer des instances
- Une définition de type
- Un espace de noms
- Une unité d'encapsulation
- Une unité de réutilisation
- Un ensemble d'instances
- Un moyen de calcul

Classe = modèle pour créer des instances

Allocation / déallocation dynamiques

- peuvent être définis par le concepteur

```
class A {
public:
    …
    void* operator new (size_t);
    void operator delete (void*);
     };
```

- variante avec [], variante pour les opérateurs new/delete non membres

Classe = modèle pour créer des instances (dans une classe)

```
class A {
 static list<A*> libres;

public:
 void* operator new (size_t s) {
  A* p;
  if (libres.empty())
   p = (A*) ::operator new (s);
  else {
   p = libres.front();
   libres.pop_front();
  }
  cout << "allocation d'un A en "
       << p << "\n";
  return p;
 }
 void operator delete (void* p) {
  A* q = (A*) p;
  libres.push_back(q);
  cout << "liberation du A en "
       << p << "\n";
 }
};

list<A*> A::libres;

A* p1 = new A();
A* p2 = new A();
delete p1;
A* p3 = new A();
```

```
allocation d'un A en 0x603010
allocation d'un A en 0x603030
liberation du A en 0x603010
allocation d'un A en 0x603010
```

Classe = modèle pour créer des instances (opérateurs globaux)

```
char mem[100000]; int ptr = 0;
// double mem[...] préférable pour les alignements d'adresse

void* operator new (size_t s) {
  int p = ptr; ptr += s;
  cout << "allocation globale de " << s << " octets a "
    << (void*) (mem + p) << "\n";
  return mem + p;
}

cout << "adresse debut allocation " << &mem << "\n";
A* p1 = new A(); A* p2 = new A(); delete p1; A* p3 = new A();
```

```
adresse debut allocation 0x602180
allocation globale de 1 octets a 0x602180
allocation globale de 1 octets a 0x602181
allocation globale de 1 octets a 0x602182
```

préférable d'avoir (sur cet exemple) un delete qui ne fait rien

Classe = modèle pour créer des instances (opérateurs avec placement)

```
void* operator new (size_t s, void* place) {
  return place;
}

int mem[100];
for (int i = 0; i < 100; i++) {
  int* p = new(mem+i) int(i);
  ...
}
```

Attention : ne pas faire delete si la place n'est pas dans le tas

- pour détruire (déconstruire) appeler directement le destructeur
- ou fournir un delete qui n'appelle pas free

Classe = définition de type

- Attributs : constituants de l'état d'une instance
- Méthodes : opérations licites sur les instances
  - accesseurs (état inchangé) → const
  - modificateurs de l'état → pas const
- Attributs et méthodes de classe (static)

Classe = définition de type
initialisation contrôlée

```
class Rational {
public:
 Rational (int n=0, int d=1) {
  if (d == 0)
      throw NullDenominator();
  if (d < 0) { n = -n; d = -d; }
  int p = gcd (abs(n), d);
  num = n / p; den = d / p;
 }
 …
 protected:
  int num; int den;
};
```

```
Rational operator*(Rational r1,
                   Rational r2) {
 return Rational(r1.num * r2.num,
                 r1.den * r2.den);
}
```

Le constructeur garantit qu'un rationnel est toujours normalisé:

2/3 * 3/4 = 1/2 (et pas 6/12)

Classe = définition de type
conversion de type

```
class Rational {
public:
 Rational (int n=0, int d=1) {
  ...
 }

 operator double() const {
  return double(num) / den ;
 }
 protected:
  int num; int den;
};
```

```
Rational F (Rational r);
double d = F (2);
// d = double(F(Rational(2,1)));
```

```
Rational operator*(Rational r1,
                   Rational r2)...
Rational r=3; // Rational r(3,1);
Rational r1 = r * 2 ;
error: ambiguous overload for
   'operator*' in 'r * 2'
candidates are:
operator*(double, int) <built-in>
Rational operator*(Rational, Rational)
```

Classe = espace de noms

Pour éviter de polluer l'espace global de nommage avec des noms locaux :

```
class Liste {
private:
 struct Chainon {
  int info;
  Chainon* suiv;
 } *tete;
public:
 …
```

```
class DoubleListe : public Liste {
private:
 struct Chainon :
            public Liste::Chainon {
  Chainon* prec;
 } *queue;
public:
 …
```

Classe = espace de noms

Pour distinguer entre synonymes:

```
class Mois {
 …
 enum Nom {janvier, février, mars,
…
};

class BarreChocolat {
 …
 enum Marque {nuts, mars, …
};
```

```
Mois::Nom m = Mois::mars;
BarreChocolat::Marque b = Barre::mars;
```

Classe = espace de noms

On peut aussi utiliser les **namespace**

```
namespace Calendrier {
  enum Mois { janvier,février,mars … };
 …
}

namespace Calendrier { // peut être répété
 …
```

```
Calendrier::Mois m;
```

```
using Calendrier;
Mois m;
```

Noms globaux dans l'espace sans nom

```
static int v;
int F (int v) {
  ::v = v + 1;
```

Classe = unité de factorisation

Dérivation publique et hiérarchie de types

```
class Figure {
 public: … void draw() const {};
};

class Circle : public Figure {
 public: … void draw() const {…}
};

class Square : public Figure {
 public: … void draw() const {…}
};

Figure* f = new Circle(…);
f->draw() // ==> Figure::draw !!!
```

Liaison statique

Classe = unité de factorisation

Polymorphisme et liaison dynamique

```
class Figure {
 public: … virtual void draw() const {…}
};

class Circle : public Figure {
 public: … void draw() const {…}
};
```

Les signatures doivent être identiques

```
Figure& rf = *new Circle(…);
  rf.draw() // Circle::draw
  Figure f = rf;
  f.draw() // Figure::draw
```

Liaison dynamique : uniquement avec accès indirect (pointeur ou référence)

Classe = ensemble d'instances

```
class A {
public:
 A (…) { … collection.insert (this);  }
 ~A() { collection.erase (this); }

 template <typename Action>
 void apply (Action a) {
  for_each (collection.begin(), collection.end(), a);
 }
 …
private:
 static set<const A*> collection;
};
```

Classe = description de calcul

```
class Factorielle {
public:
 Factorielle (int n) {
  if (n < 0) throw NonDéfinie();
  val = (n == 0 ? 1 : n * Factorielle (n - 1);
 }
 operator int() const { return val; }
private:
 int val;
}
```

```
Factorielle f3(3);
int x = f3;
int y = Factorielle(5);
```

Classe = description de calcul (objets fonctions)

```
class Cumul {
public:
 Cumul () { total = 0; }

 void operator() (int x) { total += x; }

 operator int() const { return total; }

private:
 int total;
};
```

() est le seul opérateur pouvant prendre un nombre quelconque de paramètres

```
Cumul cumuler;
for (int i = 1; i < N; ++i)
 cumuler(i);
cout << cumuler;
```

Rappels sur la construction (modes de création)

- Tout objet est construit à sa création
- objets globaux : avant le début du programme
- objets locaux : à leur point de déclaration
- paramètres et résultats par valeur : lors de leur évaluation
- attributs : à la création de l'objet qui les contient
- objets dynamiques : par l'opérateur new

Rappels sur la construction (constructions implicites)

- Toute création implique l'appel d'un constructeur
  - soit explicitement :

```
Rational r1(3,4);
Rational* p = new Rational(1,2);
```

  - ou implicitement :

```
Rational r0;
    // constructeur par défaut
Rational r2 = r0;
    // constructeur de copie
```

Rappels sur la construction (ordre de construction)

- Toute création d'objet implique la construction (éventuellement par défaut) de ses composants

1) La (ou les) classe de base
2) Les attributs propres à la classe, dans l'ordre de déclaration
3) Exécution du corps du constructeur

Rappels sur la construction (ordre de construction)

```
class A { public: A (int = 0)… }
class B { public: B (int = 0)… }
class C { public: C (int = 0)… }
```

```
class D: public A {
  public:
   A (int i, int j): c(i), A(j) {}
   …
  private:
   B b; C c;
};
```

Construction d'un D ⇒
A(j), B(0), C(i)

Rappels sur la construction (construction par défaut)

Une classe sans constructeur se voit donner un constructeur par défaut, si tous ses membres et classes parentes ont un constructeur par défaut accessible

```
class A { public: A (int = 0)… }
class B { public: B (int = 0)… }
class C { public: C (int = 0)… }

class D: public A {
 …
 private: B b; C c;
};
```

Construction d'un D : A(0), B(0), C(0)

Rappels sur la construction (construction par copie)

Le constructeur de copie est appelé

- lors d'une déclaration avec initialisation :
  `Rational r1(1,2); Rational r2 = r1;`
- lors d'un passage de paramètre par valeur ou d'un retour de fonction par valeur :

```
Rational operator* (Rational a,
                    Rational b) {
  return Rational (a.num * b.num,
                   a.den * b.den);
}
```

```
r3 = r1 * r2; // ⇒ 3 copies
```

Rappels sur la construction (construction par copie)

- Une classe sans constructeur de copie se voit donner un constructeur de copie, si tous ses membres et classes parentes ont un constructeur de copie accessible
- Idem pour l'opérateur d'affectation
- Ne pas confondre

```
Rational r2 = r1; // constr. par copie
r2 = r1;          // affectation
```

Classe = unité d'encapsulation

- Membres de A
  - privés: accessibles par toute instance de A
  - protégés: accessibles par A et descendants
  - publics: accessibles partout
- Dérivation B de A
  - privée: non privés de A ⇒ privés de B
  - protégée: publics de A ⇒ protégés de B
  - publique: accès dans B = accès dans A
- Privé est toujours le défaut

Classe = unité d'encapsulation

Accès aux membres non publics et garantie d'intégrité des objets:

```
class A {
 protected: int p;
 …
};

class B : public A {…};

class C : public B {…};

class D : public A {…};
```

```
void B::M() {
 A a; B b; C c; D d;

 p = 0;    // OK
 a.p = 0; // KO
 b.p = 0; // OK
 c.p = 0; // OK
 d.p = 0; // KO
}
```

car ni **a** ni **d** est un **B**

Classe = unité de réutilisation

Par composition

- Les composants définissent toujours leur construction
- Les droits d'accès garantissent l'intégrité des composants

```
class A {
 public: A(int);
 …
 private: int p;
};
```

```
class B {
 public: B(int);
 …
 private: A a;
};
```

```
B::B(int i)
  : a(i){}
```

```
B::B(int i)
{a.p = i;}
```

Classe = unité de réutilisation

Composition d'instances (agrégation)

- Les composants sont propres à l'agrégat
- Ils ont même durée de vie que lui
- La copie et l'affectation définies pour les composants sont, en général, suffisants pour la copie et l'affectation de l'agrégat
- Mais le type des composants est définitivement fixé

Classe = unité de réutilisation

Composition par pointeur/référence

- Les composants peuvent être partagés par plusieurs objets
- Le type effectif des composants peut être dynamique (polymorphisme)
- La durée de vie des composants est indépendante de celle du composé $\Longrightarrow$ problèmes de déallocation
- La copie et l'affectation peuvent impliquer des partages non désirés

Classe = unité de réutilisation

Par dérivation

- Les classes de base définissent toujours leur construction

```
class A {
 public:
  A(int i): p(i) {};

  void inc() { ++p; }
 …
 private: int p;
};
```

```
class B : public A {
 public:
  B(int i, int j)
                  : A(i),q(j) {}

  void inc() {
    A::inc(); ++q;
  }
  …
  private: int q;
};
```

Classe = unité de réutilisation

Dérivation publique ≡ sous-typage: un **B** est un **A**

```
class A { … };
class B: public A { … };
A* p = new B(); // polymorphisme
```

Dérivation non publique ≡ réutilisation d'implémentation: un **B** n'est pas un **A**

```
class A { … };
class B: protected A { … };
A* p = new B(); // NON
```