

# Ch. 12 - Héritage et polymorphisme

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant  
École supérieure d'Informatique



9 décembre 2020

# Table des matières

- 1 Introduction
- 2 Principes
- 3 Redéfinition et surdéfinition
- 4 Construction, destruction, affectation
- 5 Polymorphisme
- 6 Héritage multiple

# Introduction

# Overview

- Un des fondements de la POO
- Permet de « transférer » la signature d'une classe dans une autre
- En jargon C++, on parle souvent de *classe dérivée* et de *classe de base* plutôt que de sous-classe et de super-classe.

## Dérivation en C++

- Via « : » après le nom de la classe
  - `class B : A { ... };`
  - Appels super-constructeurs via la liste d'initialisation
- 
- Plusieurs « types » de dérivation possible
    - Définit la visibilité des membres de la classe de base

# Différences entre Java et C++

- En C++, il n'y a pas d'interfaces
  - Classes abstraites sans attributs
- En C++, l'héritage multiple est autorisé
  - Attention aux ambiguïtés
- Le polymorphisme n'est pas « activé » par défaut
  - Il faut dire explicitement quelles fonctions membres sont polymorphiques
  - Il faut que l'objet soit alloué dynamiquement, ou une référence
- Conversion implicite de type « tronquage » d'une classe dérivée vers une classe de base
  - Pour les objets automatiques uniquement

# Exemple

## ■ Fichier pointcol.cpp

```

1  class point
2  {
3      int x, y;
4
5      public:
6          point(int x = 0, int y = 0) : x(x), y(y) {}
7          void setLocation(int x, int y) { this->x = x; this->y = y; }
8          friend ostream& operator <<(ostream& out, const point& p)
9              { out << "(" << p.x << ", " << p.y << ")"; }
10 };
11
12 class pointcol : public point
13 {
14     short r, g, b;
15
16     public:
17         pointcol(int x, int y, short r, short g, short b) : point(x,y), r(r), g(g), b(b) {}
18 };
19
20 int main()
21 {
22     pointcol p(1,2,80,0,0); cout << p << endl;
23     p.setLocation(3,4); cout << p << endl;
24 }

```

# Principes

# Différents types de dérivation

## ■ Trois types de dérivation

1 Dérivation publique : `class B : public A { ... };`

- Les membres publics de A sont accessibles comme membres publics dans B
- Les membres protégés de A sont accessibles comme membres protégés dans B

2 Dérivation privée : `class B : private A { ... };`

- Les membres publics et protégés de A sont accessibles comme membres privés dans B

3 Dérivation protégée : `class B : protected A { ... };`

- Les membres publics et protégés de A sont accessibles comme membres protégés dans B

## ■ En l'absence de spécificateur d'accès, la dérivation est

- publique pour les `struct`
- privée pour les `class`



# Spécificateurs d'accès et amitié

- Le mot-clé `protected` permet au membre spécifié d'être accédé uniquement par le concepteur d'une sous-classe
- Résolution de portée via `::`
- Une classe dérivée n'a jamais accès aux membres privés de sa classe de base
  - Sauf si une relation d'amitié est déclarée
- Possibilité de définir un membre dans une classe à l'aide de `using`
  - Permet de changer la visibilité d'un membre dans une sous-classe
  - Éviter
- Possibilité de définir une classe dérivée comme amie d'une classe de base
  - Permet à une sous-classe d'accéder aux membres privés de sa classe de base

# Exemple 1

## ■ Fichier `specifier.cpp`

```
1  class A
2  {
3      public:
4          void printA () { cout << "A"; }
5  };
6
7  class B : public A
8  {
9      public:
10         void printB ()
11         {
12             printA (); //ok : printA is a public member of this
13             cout << "B";
14         }
15     };
16
17 int main()
18 {
19     A a; a.printA (); cout << endl;
20     B b; b.printB (); b.printA (); cout << endl; //ok : printA is a public member of B
21 }
```

# Exemple 1

## ■ Fichier specifier.cpp

```

1  class C : private A
2  {
3      public:
4          void printC ()
5          {
6              printA (); //ok : printA is a private member of this
7              cout << "C";
8          }
9  };
10
11 class D : public C
12 {
13     public:
14         void printD ()
15         {
16             //printA (); //ko : printA is a private member of superclass
17             cout << "D";
18         }
19 };
20
21 int main ()
22 {
23     C c; c.printC (); cout << endl; //c.printA (); //ko : printA is a private member of C
24     D d; d.printD (); cout << endl; //d.printA (); //ko : printA is a private member of D
25 }

```

# Exemple 1

## ■ Fichier `specifier.cpp`

```

1  class E : protected A
2  {
3      public:
4          void printE ()
5          {
6              printA (); //ok : printA is a protected member of this
7              cout << "E";
8          }
9  };
10
11 class F : public E
12 {
13     public:
14         void printF ()
15         {
16             printA (); //ok : printA is a protected member of superclass
17             cout << "F";
18         }
19 };
20
21 int main ()
22 {
23     E e; e.printE (); cout << endl; //e.printE (); //ko : printA is a protected member of E
24     F f; f.printF (); cout << endl; //f.printE (); //ko : printA is a protected member of F
25 }

```

## Exemple 2

### ■ Fichier using.cpp

```
1  class A
2  {
3      protected :
4          int i;
5      public:
6          A(int i = 0) : i(i) {}
7  };
8
9  class B : public A
10 {
11     public:
12         using A::i;
13         using A::A;
14 };
15
16 int main()
17 {
18     B b(4); cout << b.i << endl;
19     b.i = 3; cout << b.i << endl;
20 }
```

# Amitié et héritage : rappel

## 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
    - 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  $A$
  - Cf. Ch. 7

# Exemple

## ■ Fichier friend.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
13 {
14     int _j;
15
16     public:
17         B() : _j(3) {}
18         int j() const { return _j; }
19 };
```

# Exemple

## ■ Fichier friend.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     };

```



# Redéfinition et surdéfinition

## Redéfinition des membres d'une classe dérivée

- Pour redéfinir un membre d'une classe dérivée, il suffit de le déclarer avec le même prototype que celui de la classe de base.
  - Le membre de la classe de base est alors « caché ».
- Un appel au membre sur la classe dérivée appelle le membre redéfini.

### Exemple

```
■ struct A { void print() { ... } };  
■ struct B : A { void print() { ... } };  
■ B b; b.print(); //calls B::print
```

- `final` empêche la redéfinition d'un membre dans une classe dérivée, ou la dérivation d'une classe

## Accès aux membres et transtypage

- Accès aux membres via `b.print()` (dérivé) et `b.A::print()` (classe de base).
- Aucune correspondance polymorphique n'est effectuée par défaut !
  - On ne peut pas mettre un `B` automatique dans un `A` automatique.
    - Si on le fait = « tronquage des membres » (taille)
  - Si on met un `B` dynamique (pointeur) dans un `A` dynamique, on ne peut appeler que les membres de `A`.
    - Même comportement avec les références
  - Plus de détails dans la section « Polymorphisme »

# Exemple

## ■ Fichier no-poly.cpp

```

1  class point
2  {
3      protected:
4          int x, y;
5
6      public:
7          point(int a = 0, int b = 0) : x(a), y(b) {}
8
9          void print()
10         { cout << "(" << x << ", " << y << ")" << endl; }
11     };
12
13     class pointcol : public point
14     {
15         short r, g, b;
16         public:
17             pointcol(int x = 0, int y = 0, int r = 255, int g = 255, int b = 255)
18                 : point(x,y), r(r), g(g), b(b) {}
19
20             void print()
21             {
22                 cout << "(" << x << ", " << y << ") _ _ color _ _"
23                     << r << ", " << g << ", " << b << endl;
24             }
25     };

```

# Exemple

## ■ Fichier no-poly.cpp

```
1  int main()
2  {
3      point p(3,5);
4      pointcol pc (8,6,255,128,128);
5
6      p.print();
7      pc.print();
8
9      p = pc; //truncated
10     p.print(); //no polymorphism
11
12     point * ptp = &p;
13     pointcol * ptpc = &pc;
14     ptp = ptpc; //no polymorphism
15     ptp -> print();
16
17     point & rp = p;
18     pointcol & rpc = pc;
19     rp = rpc;
20     rp.print(); //no polymorphism
21 }
```

# Redéfinition et surdéfinition

## Redéfinition (overriding)

- Réécriture du prototype d'un membre d'une classe de base au sein d'une classe dérivée.

## Surdéfinition (overloading)

- Réécriture du prototype d'un membre, souvent au sein d'une même classe.
  - Seul autre cas : réécriture du prototype d'un membre `final` d'une classe de base dans une classe dérivée.
- Le compilateur effectue une résolution des liens et décide quelle fonction appeler

# Résolution des liens lors d'héritage

- Les liens sont résolus dans l'ordre suivant :
  - 1 Appel direct (correspondance exacte).
  - 2 Conversion de paramètre et appel direct avec le converti.
    - Uniquement si une conversion implicite est possible.
  - 3 Appel de base (correspondance exacte).
  - 4 Conversion de paramètre et appel de base avec le converti.
    - Uniquement si une conversion implicite est possible.

## Remarque importante

- Si on appelle une fonction membre dans une classe dérivée surdéfinissant une fonction dans la classe de base, le compilateur appelle *toujours* la fonction de la classe dérivée
  - Si ce n'est pas possible, erreur de compilation

- Comportement très différent en Java

# Exemple 1

## ■ Fichier linkres-1.cpp

```
1  class A
2  {
3      public:
4          void f(int n)
5          {
6              cout << "int_" << n << endl;
7          }
8
9          void f(char n)
10         {
11             cout << "char_" << n << endl;
12         }
13 };
14
15 class B : public A
16 {
17     public:
18         void f(float x)
19         {
20             cout << "float_" << x << endl;
21         }
22 };
```



# Exemple 1

## ■ Fichier linkres-1.cpp

```
1  int main()
2  {
3      int n = 1;
4      char c = 'a';
5      A a;
6      B b;
7
8      a.f(n);
9      a.f(c);
10     b.f(n);
11     b.f(c);
12 }
```

## Exemple 2

### ■ Fichier linkres-2.cpp

```
1  class A
2  {
3      public:
4          void f(int n)
5          {
6              cout << "A::int_" << n << endl;
7          }
8
9          void f(char n)
10         {
11             cout << "char_" << n << endl;
12         }
13 };
14
15 class B : public A
16 {
17     public:
18         void f(int n)
19         {
20             cout << "B::int_" << n << endl;
21         }
22 };
```

## Exemple 2

### ■ Fichier linkres-2.cpp

```
1  int main()
2  {
3      int n = 1;
4      char c = 'a';
5      B b;
6
7      b.f(n);
8      b.f(c);
9  }
```

## Exemple 3

### ■ Fichier linkres-3.cpp

```
1  class A
2  {
3      public:
4          void f(int n)
5          {
6              cout << "A::int_" << n << endl;
7          }
8
9          void f(char n)
10         {
11             cout << "char_" << n << endl;
12         }
13 };
14
15 class B : public A
16 {
17
18 };
```

## Exemple 3

### ■ Fichier linkres-3.cpp

```
1  int main()
2  {
3      int n = 1;
4      char c = 'a';
5      B b;
6
7      b.f(n);
8      b.f(c);
9  }
```

## Exemple 4

### ■ Fichier linkres-4.cpp

```
1  class A
2  {
3      public:
4          void f(int n)
5          {
6              cout << "A::int_" << n << endl;
7          }
8
9          void f(char n)
10         {
11             cout << "char_" << n << endl;
12         }
13 };
14
15 class B : public A
16 {
17     public:
18         void f(int n, int m)
19         {
20             cout << "int, int_" << n << "_" << m << endl;
21         }
22 };
```

## Exemple 4

### ■ Fichier linkres-4.cpp

```
1  int main()
2  {
3      int n = 1;
4      char c = 'a';
5      B b;
6
7      b.f(n);
8      b.f(c);
9  }
```

# Construction, destruction, affectation



# Ordre d'appel

- Soient  $A$  une classe de base et  $B$  une classe dérivée.
- Quand on crée/copie/déplace un objet  $B$ , on appelle, dans cet ordre :
  - 1 le constructeur de  $A$ ,
  - 2 le constructeur de  $B$ .
- Quand on détruit un objet  $B$ , on appelle, dans cet ordre :
  - le destructeur de  $B$ ,
  - le destructeur de  $A$ .
- Pour les constructeurs et destructeurs par défaut, les appels sont faits implicitement.
- Pour les constructeurs avec paramètres, il faut les appeler via la liste d'initialisation.

# Exemple

## ■ Fichier `constr-destr.cpp`

```

1  struct A
2  {
3      A() { cout << "+A()" << endl; }
4      A(int a) { cout << "+A(int)" << endl; }
5      ~A() { cout << "-A()" << endl; }
6  };
7
8  struct B : A
9  {
10     B() { cout << "+B()" << endl; }
11     B(int a, int b) : A(a) { cout << "+B(int, int)" << endl; }
12     ~B() { cout << "-B()" << endl; }
13 };
14
15 int main()
16 {
17     A a; A aa(2);
18     B b; B bb(2,2);
19 }

```

# Constructeur de recopie / déplacement

## Rappel

- Le constructeur de recopie est appelé quand
  - 1 on initialise un objet par un autre de même type (explicite),
  - 2 on passe un objet par valeur à une fonction (implicite).
- Les règles d'appel liées au constructeur sont aussi valides pour le constructeur de recopie.
- Il faut néanmoins tenir compte de certaines subtilités selon que le constructeur de recopie a été redéfini ou non.
- Même principe avec le constructeur de déplacement
- Dans les exemples suivants,  $B$  dérive publiquement de  $A$ .

## Absence de constructeur de recopie dans la classe dérivée

- Appel du constructeur de recopie par défaut de  $B$ .
  - Rappel : la recopie se fait membre à membre.
  - Les données pointées par les attributs dynamiques ne sont pas recopiées.
- La « partie » de  $B$  qui « appartient » à  $A$  est traitée comme un membre de type  $A$ .

### Règle

- Le constructeur de recopie de la classe de base est appelé implicitement.
- S'il existe, il est appelé.
- Sinon : constructeur de recopie par défaut.

# Exemple

## ■ Fichier no-recop.cpp

```

1  class point
2  {
3      protected:
4          int x, y;
5      public:
6          point(int a = 0, int b = 0) : x(a), y(b) {}
7          point(const point& p) : x(p.x), y(p.y) { cout << "+r_point" << endl; }
8          friend ostream& operator << (ostream& out, const point& p)
9          {
10             out << "(" << p.x << ", " << p.y << ")";
11         }
12     };
13
14     class pointcol : public point
15     {
16     short r, g, b;
17     public:
18         pointcol(int x = 0, int y = 0, int r = 0, int g = 0, int b = 0)
19             : point(x,y), r(r), g(g), b(b) {}
20         friend ostream& operator << (ostream& out, const pointcol& p)
21         {
22             out << "(" << p.x << ", " << p.y << ")_color_"
23                 << p.r << " " << p.g << " " << p.b;
24         }
25     };

```

# Exemple

## ■ Fichier no-recop.cpp

```
1 void f(pointcol p)
2 {
3     cout << "f" << endl;
4 }
5
6 int main()
7 {
8     pointcol a(1,2,3);
9     f(a);
10 }
```

# Présence de constructeur de recopie dans la classe dérivée

- Il est nécessaire de recopier la partie de la classe de base  $A$ .
- Dans ce cas-ci, la recopie est explicite.

## Règle

- Le constructeur de recopie de la classe dérivée doit prendre en charge *l'intégralité* de la recopie de l'objet.
- En l'occurrence, pas uniquement sa partie dérivée.
- En général, on recommande d'appeler le constructeur de recopie de la classe de base via la liste d'initialisation.
  - S'il existe, il est appelé.
  - Sinon : constructeur de recopie par défaut.

# Exemple

## ■ Fichier `recop.cpp`

```

1  class point
2  {
3      protected:
4          int x, y;
5
6      public:
7          point(int a = 0, int b = 0) : x(a), y(b) {}
8
9          point(const point& p) : x(p.x), y(p.y)
10         {
11             cout << "+r_point" << endl;
12         }
13
14         friend ostream& operator << (ostream& out, const point& p)
15         {
16             out << "(" << p.x << ", " << p.y << ")";
17         }
18     };

```



# Exemple

## ■ Fichier `recop.cpp`

```

1  class pointcol : public point
2  {
3      short r, g, b;
4
5      public:
6          pointcol(int x = 0, int y = 0, int r = 0, int g = 0, int b = 0)
7              : point(x,y), r(r), g(g), b(b) {}
8
9          pointcol(const pointcol &p) : point(p), r(p.r), g(p.g), b(p.g)
10         {
11             cout << "+r_pointcol" << endl;
12         }
13
14         friend ostream& operator << (ostream& out, const pointcol& p)
15         {
16             out << "(" << p.x << ", " << p.y << ")_color_"
17                 << p.r << " " << p.g << " " << p.b;
18         }
19     };

```

# Exemple

## ■ Fichier `recop.cpp`

```
1 void f(pointcol p)
2 {
3     cout << "f" << endl;
4 }
5
6 int main()
7 {
8     pointcol a(1,2,255,128,128);
9     f(a);
10 }
```

# Exemple d'erreur

## ■ Fichier `recop-error.cpp`

```
1 class point
2 {
3     protected:
4         int x, y;
5
6     public:
7         point(int a = 0, int b = 0) : x(a), y(b) {}
8
9         friend ostream& operator << (ostream& out, const point& p)
10        {
11            out << "(" << p.x << ", " << p.y << ")";
12        }
13    };
```

# Exemple d'erreur

## ■ Fichier `recop-error.cpp`

```

1  class pointcol : public point
2  {
3      short r, g, b;
4
5      public:
6          pointcol(int x = 0, int y = 0, int r = 0, int g = 0, int b = 0)
7              : point(x,y), r(r), g(g), b(b) {}
8
9          pointcol(const pointcol &p) : r(p.r), g(p.g), b(p.g) //no call to point(p)
10         {
11             cout << "+r_pointcol" << endl;
12         }
13
14         friend ostream& operator << (ostream& out, const pointcol& p)
15         {
16             out << "(" << p.x << ", " << p.y << ")_color_"
17                 << p.r << " " << p.g << " " << p.b;
18         }
19     };

```

# Exemple d'erreur

## ■ Fichier `recop-error.cpp`

```
1 void f(pointcol p)
2 {
3     cout << "f_␣" << p << endl;
4 }
5
6 int main()
7 {
8     pointcol a(1,2,255,128,128);
9     f(a);
10 }
```

## Absence d'opérateur d'affectation (copie / mouvement) dans la classe dérivée

- L'opérateur d'affectation (copie / mouvement) peut être surdéfini dans toute classe (en particulier, dans une classe de base  $A$ ).
- L'affectation (copie / mouvement) est effectuée membre à membre, les données pointées par les attributs dynamiques ne sont pas recopiées.

### Règle

- L'opérateur d'affectation (copie / mouvement) de la classe de base est appelé implicitement.
- S'il existe, il est appelé.
- Sinon : affectation (copie / mouvement) par défaut.

# Exemple

## ■ Fichier no-affect.cpp

```

1  class point
2  {
3      protected:
4          int x, y;
5
6      public:
7          point(int a = 0, int b = 0) : x(a), y(b) {}
8
9          point & operator =(const point& p)
10         {
11             if (this != &p)
12             {
13                 x = p.x;
14                 y = p.y;
15                 cout << "=_point" << endl;
16             }
17             return *this;
18         }
19
20         friend ostream& operator << (ostream& out, const point& p)
21         {
22             out << "(" << p.x << ", " << p.y << ")";
23         }
24     };

```

# Exemple

## ■ Fichier no-affect.cpp

```

1  class pointcol : public point
2  {
3      short r, g, b;
4
5      public:
6          pointcol(int x = 0, int y = 0, int r = 0, int g = 0, int b = 0)
7              : point(x,y), r(r), g(g), b(b) {}
8
9          friend ostream& operator << (ostream& out, const pointcol& p)
10         {
11             out << "(" << p.x << ", " << p.y << ", " << p.r << ", " << p.g << ", " << p.b << ")" << endl;
12         }
13     };
14
15     int main()
16     {
17         pointcol p1(1,2,255, 128, 128); pointcol p2(4,5,255, 128, 128);
18         p2 = p1; cout << p1 << endl; cout << p2 << endl;
19
20         pointcol * pt1 = new pointcol(1,2,255,128,128);
21         pointcol * pt2 = new pointcol(4,5,255,128,128);
22         pt1 = pt2; cout << *pt1 << endl; cout << *pt2 << endl;
23     }
24 
```



## Présence d'opérateur d'affectation (copie / mouvement) dans la classe dérivée

- Il est nécessaire d'affecter la partie de la classe de base.
- Cette affectation (copie / mouvement) doit être faite explicitement, comme pour le constructeur de recopie.

### Règle

- L'opérateur d'affectation (copie / mouvement) dans la classe dérivée doit prendre en charge l'intégralité de l'affectation de l'objet.
- Changement par rapport au constructeur de recopie : pas de liste d'initialisation.
- Il faut faire autrement.

# Exemple

## ■ Fichier affect.cpp

```

1  class point
2  {
3      protected:
4          int x, y;
5
6      public:
7          point(int a = 0, int b = 0) : x(a), y(b) {}
8
9          point & operator =(const point& p)
10         {
11             if (this != &p)
12             {
13                 x = p.x;
14                 y = p.y;
15                 cout << "=_point" << endl;
16             }
17             return *this;
18         }
19
20         friend ostream& operator << (ostream& out, const point& p)
21         {
22             out << "(" << p.x << ", " << p.y << ")";
23         }
24     };

```

# Exemple

## ■ Fichier affect.cpp

```

1  class pointcol : public point
2  {
3      short r, g, b;
4      public:
5          pointcol(int x = 0, int y = 0, int r = 0, int g = 0, int b = 0)
6              : point(x,y), r(r), g(g), b(b) {}
7          pointcol & operator =(const pointcol& p)
8          {
9              if(this != &p)
10             {
11                 point * p1 = this; //we want the same types without truncating
12                 const point* p2 = &p; //we want the same types without truncating, const allows =
13                 *p1 = *p2; //affectation in point
14                 r = p.r; g = p.g; b = p.b;
15                 cout << "pointcol" << endl;
16             }
17             return *this;
18         }
19     friend ostream& operator << (ostream& out, const pointcol& p)
20     {
21         out << "(" << p.x << ", " << p.y << ", " << "color"
22             << p.r << ", " << p.g << ", " << p.b;
23     }
24 };

```

# Exemple

## ■ Fichier affect2.cpp

```

1  class pointcol : public point
2  {
3      short r, g, b;
4      public:
5          pointcol(int x = 0, int y = 0, int r = 0, int g = 0, int b = 0)
6              : point(x,y), r(r), g(g), b(b) {}
7          pointcol & operator =(const pointcol& p)
8          {
9              if(this != &p)
10             {
11                 point & p1 = *this; //we want the same types without truncating
12                 const point& p2 = p; //we want the same types without truncating , const allows =
13                 p1 = p2; //affectation in point
14                 r = p.r; g = p.g; b = p.b;
15                 cout << "pointcol" << endl;
16             }
17             return *this;
18         }
19     friend ostream& operator << (ostream& out, const pointcol& p)
20     {
21         out << "(" << p.x << ", " << p.y << ", " << p.r << ", " << p.g << ", " << p.b << ")";
22         out << "color: " << p.r << ", " << p.g << ", " << p.b << "\n";
23     }
24 };

```

# Exemple

## ■ Fichier affect.cpp

```
1  int main()
2  {
3      pointcol p1(1,2,255, 128, 128);
4      pointcol p2(4,5,255, 128, 128);
5
6      cout << p1 << endl;
7      cout << p2 << endl;
8      cout << endl;
9
10     p2 = p1;
11     cout << p1 << endl;
12     cout << p2 << endl;
13
14     pointcol * pt1 = new pointcol(1,2,255,128,128);
15     pointcol * pt2 = new pointcol(4,5,255,128,128);
16
17     delete pt1;
18
19     pt1 = pt2;
20     cout << *pt1 << endl;
21     cout << *pt2 << endl;
22
23     delete pt1;
24     delete pt2;
25 }
```

# Polymorphisme

# Introduction

- Conceptuellement, si  $B$  hérite de  $A$ , tous les objets de  $B$  sont des objets de  $A$ .
  - Toutes les voitures sont des véhicules, mais tous les véhicules ne sont pas des voitures.
- Cette compatibilité apparaît en  $C++$  dans le cas de dérivation publique.
- Certaines conversions implicites sont autorisées :
  - objet dérivé en objet de base (avec tronquage possible),
  - d'un pointeur (resp. référence) sur une classe dérivée en un pointeur (resp. référence) sur une classe de base.

# Conversion automatique d'un type dérivé en type de base

- Un objet dérivé est souvent plus gros qu'un objet de base
- Avec les objets automatiques, on manipule de « vraies » données
  - Pas des adresses ou des alias

## Résultat

- Les objets convertis sont « tronqués »
- Les données « spécifiques » aux type dérivés sont perdues
- Les conversions dans l'autre sens provoquent une erreur à la compilation
  - Quel que soit le type de conversion
  - Affectation, `static_cast`, `dynamic_cast`



# Conversion dynamique d'un type dérivé en type de base

- Ici, on ne manipule pas des données, mais des adresses ou des alias
- Le problème de taille précédent n'est plus présent ici

## Résultat

- Les objets convertis sont cohérents
- Les conversions dans l'autre sens provoquent des résultats variés
  - Résultats incohérents, erreur de segmentation, `bad_cast`, `nullptr`

# Exemple

## ■ Fichier conv.cpp

```

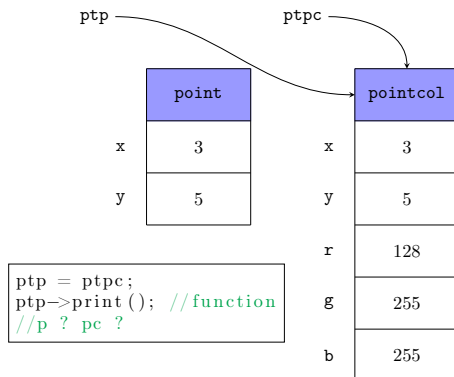
1 point p (1,2); pointcol pc (3,4,128,255,255);
2 p = pc; //pp truncated : p is "really" a point
3 //pc = p; // ko
4 //pc = static_cast<pointcol>(p); //ko
5
6 p = point(1,2); point& rp = p; pc = pointcol(3,4,128,255,255); pointcol& rpc = pc;
7 rp = rpc; //no truncation
8
9 p = point(1,2);
10 //rpc = rp; //ko
11 rpc = static_cast<pointcol&>(rp); // ok, but incoherent result
12 //rpc = dynamic_cast<pointcol&>(rp); //ok, launches bad_cast
13
14 point * ptp = new point(1,2); pointcol * ptpc = new pointcol(3,4,128,255,255);
15 ptp = ptpc; //no truncation
16
17 ptp = new point(1,2);
18 //ptpc = ptp; //ko
19 //ptpc = static_cast<pointcol*>(ptp); //ok, but seg fault
20 if (pointcol * converted = dynamic_cast<pointcol*>(ptp))
21 {
22     cout << (*ptp) << endl;
23     cout << (*ptpc) << endl;
24 }
25 else
26     cout << "You_cannot_convert_this_point_to_a_pointcol" << endl;

```

# Résolution statique des liens : illustration

## Question : fichier `static.cpp`

- Que font les instructions suivantes ?



# Résolution statique des liens

- `ptp` est de type `point` mais l'objet pointé par `ptp` est de type `pointcol`.
  - On choisit donc la fonction `print` de `point`

## Résolution statique des liens

- Le choix de la fonction est effectué à la compilation, une fois pour toutes
- Intuitivement, le type des objets pointés (ou non) est décidé et figé à la compilation.
- Un choix de résolution ne peut-être effectué qu'avec des pointeurs et des références
  - Avec des objets automatiques, un tronquage aurait été effectué
  - Comportement indéterminé

# Résolution dynamique des liens

- La fonction de la classe « la plus profonde » doit être appelée.

## Résolution dynamique des liens

- La fonction à appeler doit être déterminée à l'exécution.
- Possible via des objets dynamiques et des méthodes virtuelles.
  - Mot-clé `virtual`
- Déclaration (suffisant) dans la classe de base.
- Intuitivement : « tous les enfants définissent la fonction comme ils l'entendent ».

# Exemple

## ■ Fichier dynamic.cpp

```

1  class point
2  {
3      protected:
4          int x, y;
5      public:
6          point(int a = 0, int b = 0) : x(a), y(b) {}
7
8          virtual void print() { cout << "(" << x << ", " << y << ")" << endl; }
9  };
10
11  class pointcol : public point
12  {
13      short r, g, b;
14
15      public:
16          pointcol(int x = 0, int y = 0, int r = 255, int g = 255, int b = 255)
17              : point(x,y), r(r), g(g), b(b) {}
18
19          void print()
20          {
21              cout << "(" << x << ", " << y << ")"
22                  << " color " << r << " " << g << " " << b << endl;
23          }
24  };

```

# Résolution dynamique des liens et polymorphisme

- Par défaut, toutes les résolutions de liens sont statiques.
  - Performance
- `virtual` impose une résolution dynamique des liens pour la fonction ainsi que toutes ses redéfinitions.
  - Légère perte de performance à l'exécution (runtime).
- On peut également faire de la détermination de type à l'exécution avec des objets dynamiques, sans tronquage.
- Résolution dynamique des liens + résolution dynamique des types = polymorphisme.

## Remarque

- Le polymorphisme ne s'applique jamais aux paramètres

# Exemple

## ■ Fichier `params.cpp`

```
1 struct A
2 {
3     virtual void f(A&) { cout << "A::f(A)" << endl; }
4 };
5
6 struct B : A
7 {
8     virtual void f(A&) { cout << "B::f(A)" << endl; }
9     virtual void f(B&) { cout << "B::f(B)" << endl; }
10 };
11
12 int main()
13 {
14     A a; B b; A& ra = a;
15     ra.f(a); ra.f(b);
16
17     ra = b; ra.f(a); ra.f(b);
18
19     A& rab = b; rab.f(a); rab.f(b);
20
21     A * pa = new A; B * pb = new B;
22     pa->f(*pa); pa->f(*pb);
23
24     pa = pb; pa->f(*pa); pa->f(*pb);
25 }
```



# Debriefing

- $B : : f(b)$  n'est *jamais* appelé
  - Car c'est une surdéfinition, pas une redéfinition

## Rappel

- Les références sont *constantes*
- Réaffecter une référence ne change pas la référence, mais l'objet référencé
- Le type de l'objet référencé est déterminé dynamiquement (à l'exécution) lors de son initialisation
  - Impossible de le changer après
- Ce genre de comportement « ne se produit pas » avec des pointeurs
  - Émulation possible avec pointeurs constants

# Le mot-clé `override`

- On peut préciser qu'une fonction dérivée redéfinit une fonction de base
- Mot-clé `override`
  - Ex. : `void f() override { ... }`
- Facultatif, mais si on l'utilise pas, erreur de compilation
  - Par ex., sur une surdéfinition

## Hygiène de programmation

- Utilisez le mot-clé `override` à chaque fois que vous le pouvez

# Opérateur d'affectation

- La surcharge d'opérateur peut être virtuelle.
- L'affectation en particulier
  - ... mais elle se comporte différemment d'une fonction « habituelle ».
- Redéfinir l'opérateur d'affectation dans une classe dérivée ne redéfinit pas l'opérateur d'affectation dans une classe de base.
  - Parce que c'est une surdéfinition (paramètres différents)

## Conclusion

- Le « polymorphisme » ne s'applique pas en cas d'affectation.

# Exemple

## ■ Fichier affect-virtual.cpp

```
1  class A
2  {
3      public:
4          virtual A & operator = (const A&) { cout << "=A" << endl; }
5  };
6
7  class B : public A
8  {
9      public:
10         virtual B & operator = (const B&) //override //ko
11         { cout << "=B" << endl; }
12     };
13
14     int main()
15     {
16         A * a1 = new A; A * a2 = new A;
17         B * b1 = new B; B * b2 = new B;
18
19         *b1 = *b2;
20         *a1 = *b1;
21         *a1 = *b2;
22     }
```

## ■ Opérateur surdéfini, pas redéfini

# Propriétés des fonctions virtuelles

- Une fonction virtuelle dans une classe l'est dans toutes ses classes dérivées.
- Redéfinir une fonction virtuelle n'est pas obligatoire.
- On peut surdéfinir ( $\neq$  redéfinir) une fonction virtuelle.
  - ... mais c'est d'intérêt discutable.
- Le type de retour  $R$  ne peut pas être changé.
  - ... sauf si on retourne un type dérivé de  $R$
- Seule une fonction membre peut être virtuelle.
- Un constructeur ne peut pas être virtuel.
  - ... mais un destructeur, si.
- Attention : opérateur d'affectation.
  - Pas de polymorphisme sur les paramètres

# Destructeur virtuel

## Hygiène de programmation

- Dans une classe de base polymorphe, prévoir soit
  - aucun destructeur,
  - un destructeur privé ou protégé,
  - un destructeur public virtuel.

## Motivation

- Pour éviter une résolution statique des liens, soit on la « rend dynamique », soit on empêche la destruction.
- On veut absolument appeler les destructeurs des sous-classes
  - Idée : « éviter les ennuis »

# Illustration

## ■ Fichier destr-virt.cpp

```
1 struct Mere {  
2     ~Mere() { cout << "-M" << endl; }  
3     virtual void print() { cout << "Mère_" << endl; }  
4 };  
5  
6 struct Fille : Mere {  
7     ~Fille() { cout << "-F" << endl; }  
8     void print() { cout << "Fille_" << endl; }  
9 };  
10  
11 void print_indep(Mere* m) { m->print(); }  
12  
13 int main() {  
14     Mere * m = new Fille;  
15     print_indep(m);  
16     delete m;  
17 }
```

# Fonctions virtuelles pures

- Idée : créer des classes *abstraites* qui ne serviront qu'à être dérivées.
- On peut forcer la redéfinition de certaines fonctions dont on ne connaît *a priori* pas le comportement.
  - Exemple : `sort` d'un algorithme de tri abstrait qui peut être implémenter par insertion, en bulle, etc.
- À l'évidence, ces classes ne peuvent être instanciées.
- Utilisation de *fonctions virtuelles pures*
  - `virtual void sort() = 0;`
- Définition *nulle*, pas *vide*.



# Classes abstraites

- Une classe comportant une fonction virtuelle pure est considérée comme *abstraite*.
- Une classe abstraite ne peut pas être instanciée
  - ... mais on peut toujours avoir des objets dynamiques
  - ... et des déclarations d'objets dynamiques.
- Motivation : polymorphisme
- Une fonction virtuelle pure dans une classe de base doit obligatoirement être soit
  - redéfinie dans les classes dérivées,
  - déclarée à nouveau explicitement comme virtuelle pure dans les classes dérivées.

# Exemple

## ■ Fichier `abstract.cpp`

```
1  struct vehicle
2  {
3      virtual int nbWheels() = 0;
4  };
5
6  struct car : vehicle
7  {
8      int nbWheels() override { return 4; }
9  };
10
11 struct bike : vehicle
12 {
13     int nbWheels() override { return 2; }
14 };
```

# Exemple

## ■ Fichier abstract.cpp

```

1  int main()
2  {
3      car c;
4      bike b;
5
6      cout << "Bike_with_" << b.nbWheels() << "_wheels" << endl;
7      cout << "Car_with_" << c.nbWheels() << "_wheels" << endl;
8
9      //vehicle v = bike(); //KO (instance)
10     //vehicle v; //KO (instance)
11
12     vehicle & rv = b; //ok
13     cout << "Vehicle_with_" << rv.nbWheels() << "_wheels" << endl;
14     rv = c; //reaffacting a ref does not change the ref, but the referenced object
15     cout << "Vehicle_with_" << rv.nbWheels() << "_wheels" << endl;
16
17     vehicle * v = new car();
18     cout << "Vehicle_with_" << v -> nbWheels() << "_wheels" << endl;
19 }

```

# Héritage multiple

# Introduction

- Concrètement, permet à une classe d'être dérivée de plusieurs classes.
  - Rappel : pas d'interfaces en C++.
- Les règles vues pour l'héritage « simple » sont d'application pour l'héritage multiple.
- Mêmes « problèmes » liés à la résolution statique des liens, à la présence ou non de constructeurs de copie, à la surcharge de l'affectation, etc.
- Mêmes règles d'appel de constructeurs et de destructeurs.
  - Quelques particularités toutefois.
- Mêmes manières d'appeler les fonctions et constructeurs des classes de base.

# Exemple

## ■ Fichier `multiple.cpp`

```
1  class point
2  {
3      int x, y;
4
5      public:
6          point(int a = 0, int b = 0) : x(a), y(b) {}
7
8          virtual void print()
9          {
10             cout << "(" << x << ", " << y << ")";
11         }
12     };
13
14     class color
15     {
16         short r, g, b;
17
18         public:
19             color(int r = 0, int g = 0, int b = 0) : r(r), g(g), b(b) {}
20
21             virtual void print()
22             {
23                 cout << "[" << r << ", " << g << ", " << b << "]";
24             }
25     };
```

# Exemple

## ■ Fichier multiple.cpp

```

1  class pointcol : public point, public color
2  {
3      public:
4          pointcol(int x = 0, int y = 0, int r = 0, int g = 0, int b = 0)
5              : point(x,y), color(r,g,b) {}
6
7          void print() override
8          {
9              cout << "{_";
10             point::print(); cout << "_,_"; color::print();
11             cout << "_}";
12         }
13 };
14
15 int main()
16 {
17     pointcol p(1,2,100,128,255);
18     p.print(); cout << endl;
19     p.point::print(); cout << endl;
20     p.color::print(); cout << endl;
21
22     color & c = p; c.print(); cout << endl;
23     point& pp = p; pp.print(); cout << endl;
24 }

```

# Appel des constructeurs et destructeurs

## Héritage simple

- Constructeurs : ordre de dérivation (base > dérivée).
- Destructeurs : ordre inverse de dérivation (dérivée > base).

## Héritage multiple

- Constructeurs : ordre de dérivation, par ordre de déclaration (base1 > base2 > dérivée).
- Destructeurs : ordre inverse de dérivation, par ordre inverse de déclaration (dérivée > base2 > base1).



## ■ Fichier cstr-mult.cpp

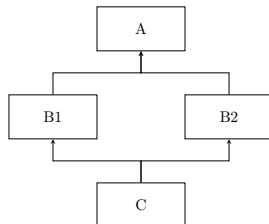
```

1 struct A
2 {
3     A() { cout << "+A" << endl; }
4     A(const A& a) { cout << "rA" << endl; }
5     virtual ~A() { cout << "-A" << endl; }
6 };
7
8 struct B
9 {
10     B() { cout << "+B" << endl; }
11     B(const B& a) { cout << "rB" << endl; }
12     virtual ~B() { cout << "-B" << endl; }
13 };
14
15 struct C : A, B
16 {
17     C() { cout << "+C" << endl; }
18     C(const C& a) { cout << "rC" << endl; }
19     virtual ~C() { cout << "-C" << endl; }
20 };
21
22 void f(A a) {}
23
24 int main()
25 {
26     C c; cout << endl;
27     f(c); cout << endl;
28     C * cc = new C(); cout << endl;
29     delete cc; cout << endl;
30 }

```

## Problème lié à l'héritage multiple

- Supposez qu'on ait le schéma de dérivation suivant



### Question

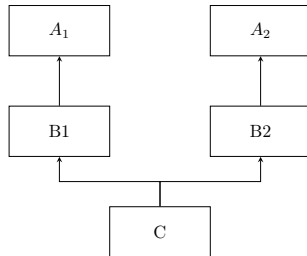
- Que fait un appel depuis C à un attribut de A, si B1 et B2 ne construisent pas A de la même façon ?

## ■ Fichier diamond.cpp

```
1  struct A
2  {
3      int i;
4      A(int i = 0) : i(i) {}
5  };
6
7  struct B1 : A
8  {
9      B1(int j = 0) : A(j) {}
10 };
11
12 struct B2 : A
13 {
14     B2(int j = 0) : A(j) {}
15 };
16
17 struct C : B1, B2
18 {
19     C(int j1 = 0, int j2 = 0) : B1(j1), B2(j2) {}
20 };
21
22 int main()
23 {
24     C c (2, 4);
25     cout << c.i << endl; //?!
26 }
```

## Diamond of DEATH : le problème

- On remarque que la classe  $C$  possède deux copies de « super-objets » de types  $A$
- Ces super-objets sont distincts, avec des attributs distincts



- L'appel  $c.i$  est ambigu.
  - On ne sait pas quel « chemin de dérivation » privilégier.
- Ordre d'appel des constructeurs :  $A_1$   $B1$   $A_2$   $B2$   $C$

## Dérivation « virtuelle »

- L'héritage multiple peut conduire à une duplication des membres.
  - Attributs : problématique.
  - Fonctions : moins d'importance (résolution de portée).
- En général, on ne veut délibérément une duplication des attributs
  - Bonne pratique ? Synchronisation ?
- On pourrait ne travailler qu'avec un jeu de données
  - Pertinence ? Synchronisation ?

### Solution : « dérivation virtuelle »

- `class B1 : public virtual A`
- `class B2 : public virtual A`
- `class C : public B1, public B2`

# Construction : transmission des arguments

## Problème

- Quels arguments transmettre au constructeur ? Ceux de  $B1$  ou ceux de  $B2$  ?

## Solution

- En cas de dérivation virtuelle *uniquement*, on peut spécifier dans  $C$  des informations destinées à  $A$ .
- Exemple :  
`C(int j1, int j2) : A((j1 + j2) / 2, ... {}`

## ■ Fichier diamond2.cpp

```
1  struct A
2  {
3      int i;
4      A(int i = 0) : i(i) {}
5  };
6
7  struct B1 : A
8  {
9      B1(int j = 0) : A(j) {}
10 };
11
12 struct B2 : A
13 {
14     B2(int j = 0) : A(j) {}
15 };
16
17 struct C : B1, B2
18 {
19     C(int j1 = 0, int j2 = 0) : A((j1 + j2) / 2, B1(j1), B2(j2)) {}
20 };
21
22 int main()
23 {
24     C c (2, 4);
25     cout << c.i << endl;
26     cout << c.B1::i << endl;
27     cout << c.B2::i << endl;
28 }
```

## Remarque

- Ordre d'appel : le constructeur d'une classe virtuelle est toujours appelé avant les autres.
- Exemple précédent :  $A \ B1 \ B2 \ C$ .

## Hygiène de programmation

- Hériter « comme en Java »
  - Ne pas avoir de « diamant »,
  - Ne pas hériter multiples fois s'il y a des attributs en commun
- Motivation : pouvoir instancier « correctement et pertinemment » les classes de type  $B1$  ou  $B2$ .