MOOC semaine 6
Héritage multiple

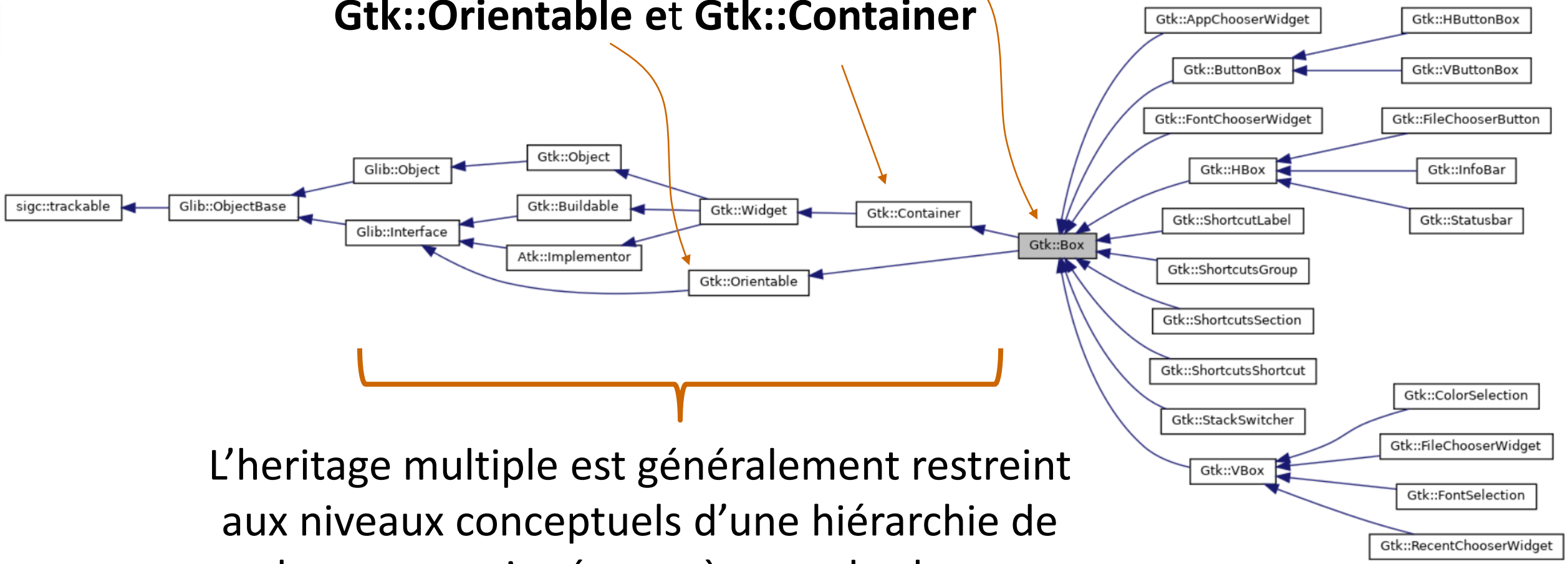
Objectif: Définir clairement des concepts (ou propriétés) *indépendants* dans des classes et s'en servir comme un menu pour créer des classes plus élaborées par héritage multiple.

Plan:

- Brève illustration sur Gtk
- Le lien virtuel: un second usage pour le mot clef **virtual**
- Exemples: configuration en losange avec/sans lien virtuels

Exemple: la classe **Gtk::Box**
hérite des classes
Gtk::Orientable et **Gtk::Container**

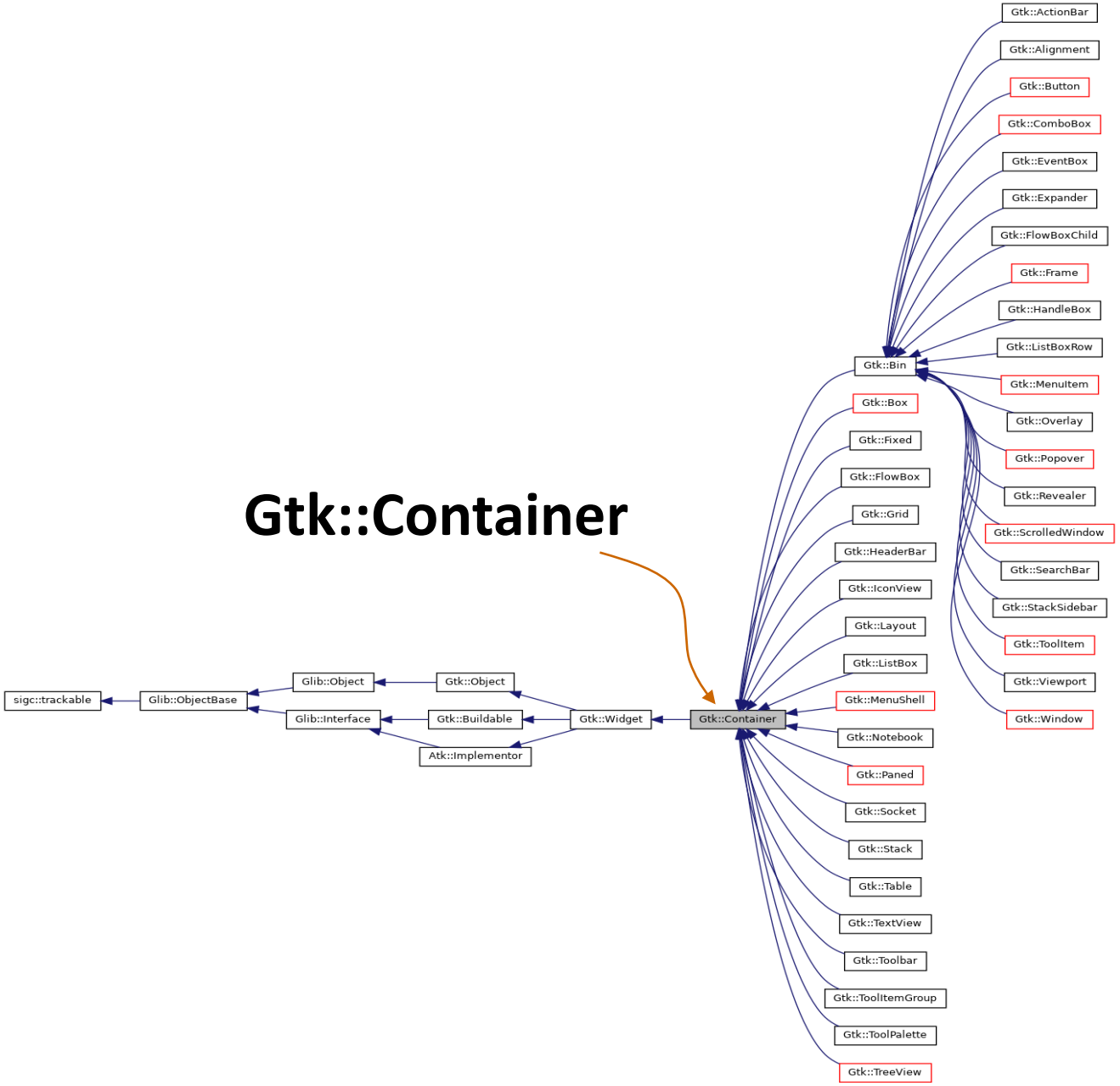
Spécialisation



L'heritage multiple est généralement restreint aux niveaux conceptuels d'une hiérarchie de classe et pratiqué sur très peu de classes

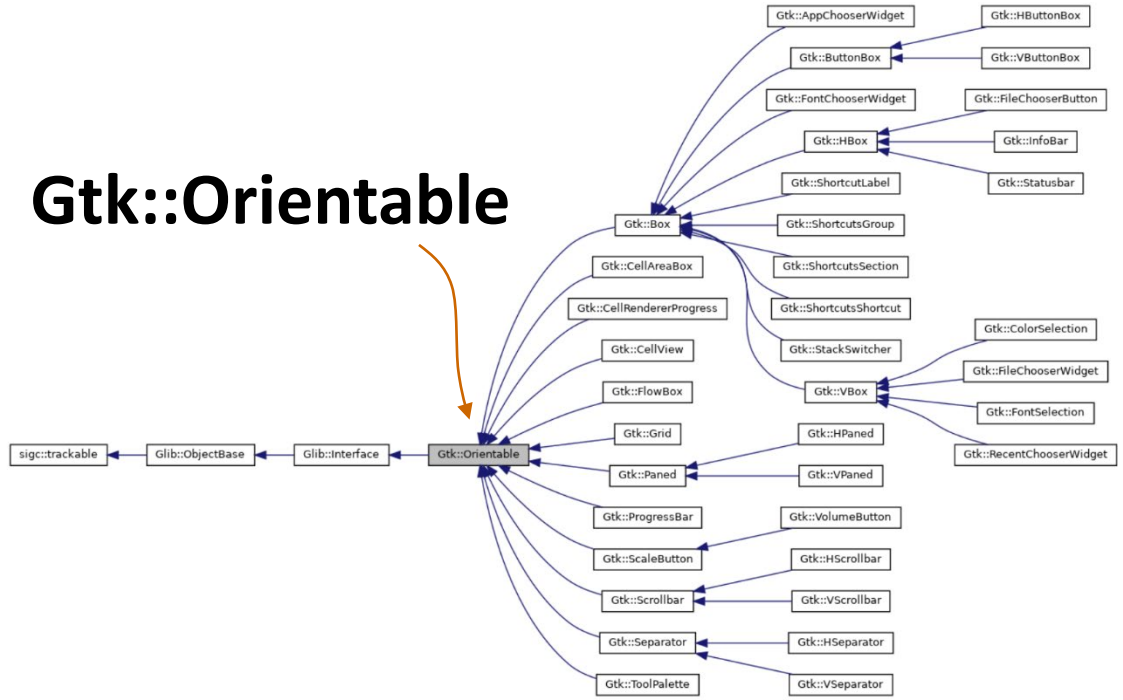
Spécialisation

Gtk::Container



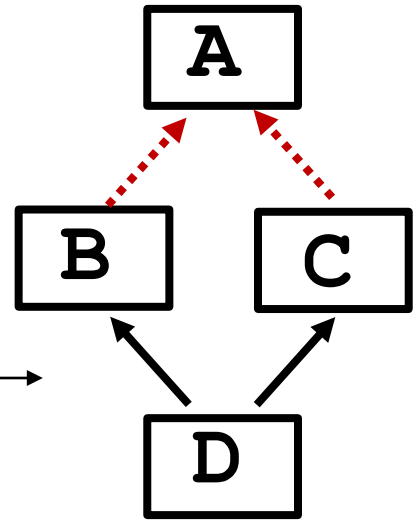
Spécialisation

Gtk::Orientable



EPFL

Déclaration de l'héritage multiple



Indiquer à la déclaration la liste des classes dont on hérite:

Exemple: `class D : public B, public C { };`

L'indication d'un *lien* **`virtual`** est généralement nécessaire **au niveau supérieur des classes parentes (ici B et C)** pour éviter la duplication des attributs provenant de la classe A :

Exemple:

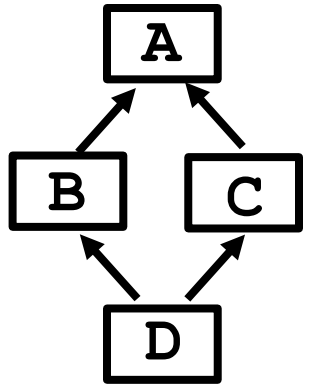
```
class B : public virtual A {};
class C : public virtual A {};
```

Points à surveiller: ordre des classe => ordre des constructeurs, constructeur par défaut

sans_virtual

```
class A
{
public:
  A():a(0) {}
  int obsene;
private:
  int a;
};

class B : public A
{
};

class C : public A
{
};

class D : public B, public C
{
};

int main()
{
  D d1;
  cout << d1.B::obsene << endl;
  cout << d1.C::obsene << endl;
}
```

Cet exemple syntaxiquement correct, **sans lien `virtual`**, illustre la **duplication de l'attribut** dans la classe D



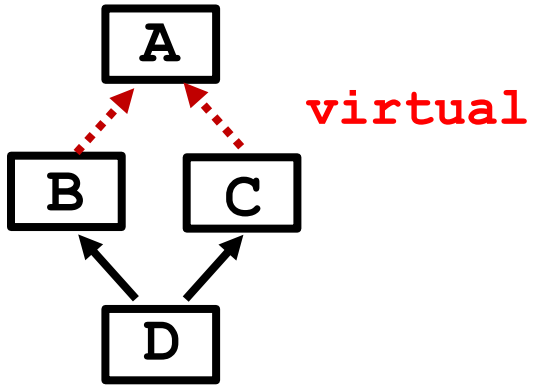
Affichage possible à l'exécution:

```
4196832
4196240
```

*Rappel: ne **PAS** rendre des attributs public*

```
class A
{
public:
  A() : a(33) {}
  A(int x): a(x){}
  void afficher(){cout << a << endl;}
private:
  int a;
};

class B : public virtual A
{
public:
  B() : A(1) {}
};

class C : public virtual A
{
public:
  C() : A(2) {}
};

class D : public B, public C {};

int main()
{
  D d1;
  d1.afficher();
  return 0;
}
```

avec virtual



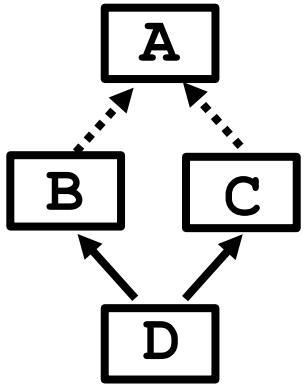
Examinons la déclaration de **d1** dans **main()**:

- Aucune valeur initiale fournie
- Constructeur par défaut par défaut de D
- Du fait des liens **virtual** entre les classes **B** et **C** et leur supeclasse **A**, => **appel explicite du constructeur par défaut de A**, avant l'appel par défaut des constructeurs de B puis de C.
- Pour la même raison, **PAS d'appels aux constructeurs de A depuis les constructeurs par défaut de B et de C**

Compile et affiche 33 à l'exécution

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

class B : public virtual A
{
public:
  B() : A(0) {}
};

class C : public virtual A
{
public:
  C() : A(1) {}
};

class D : public B, public C
{
};

int main()
{
  D d1;
  return 0;
}
```

Quizz1



Question1 : ce code…

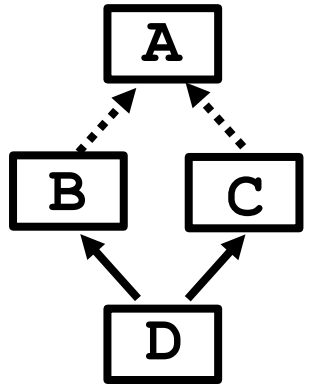
A compile et s'exécute sans problème (il ne fait rien, le programme se termine immédiatement)

B compile mais exécution avec segmentation fault

C ne compile pas

```
…
class A
{
public:
  void f()const { cout << "Arg!"; }
};

class B : public virtual A
{ };

class C : public virtual A
{ };

class D : public B, public C
{ };

int main()
{
  D d1;
  d1.f();
  return 0;
}
```

Quizz2

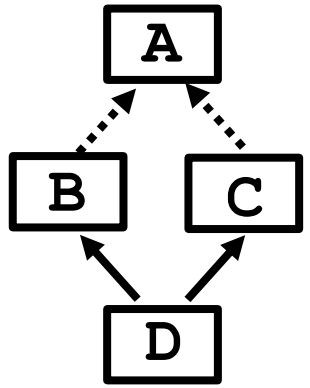


Question2 : ce code …

A ne compile pas
B compile et n'affiche rien
C compile et affiche **`Arg!`**
D compile et affiche **`Arg!Arg!`**

```
…
class A
{
public:
  void f() const { cout << "A "; }
};

class B : public virtual A
{
public:
  void f() const { cout << "B "; }
};

class C : public virtual A
{
public:
  void f() const { cout << "C "; }
};

class D : public B, public C { };

int main()
{
  D d1;
  d1.f();
  return 0;
}
```

Quizz3

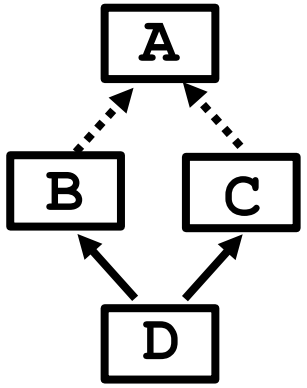


Question3 : ce code …

A ne compile pas
B compile et affiche **ABC**
C compile et affiche **AB**
D compile et affiche **AC**
E compile et affiche **BC**
F compile et affiche **A**

Exemple 4 avec virtual et using

```
class A
{
public:
  void f() const { cout << "A "; }
};

class B : public virtual A
{
public:
  void f() const { cout << "B "; }
};

class C : public virtual A
{
public:
  void f() const { cout << "C "; }
};

class D : public B, public C
{ public: using C::f; };

int main()
{
  D d1;
  d1.f();
  return 0;
}
```



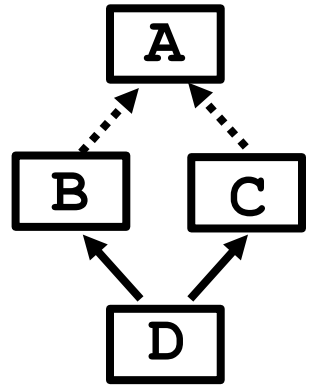
Examinons l'usage de **using** :

- La classe **D** lève l'ambiguïté sur la méthode **f** à utiliser pour ses instances en indiquant qu'il faut utiliser la méthode de la **classe C**
- C'est ce qui est fait pour l'instance d1 : l'exécution affiche C

Exemple 5

```
class A
{
public:
  virtual void f() const { cout << "A "; }
};

class B : public virtual A
{
public:
  void f() const { cout << "B "; }
};

class C : public virtual A
{
public:
  void f() const { cout << "C "; }
};

class D : public B, public C
{ public: using C::f; };

int main()
{
  return 0;
}
```

avec virtual pour méthode, virtual pour héritage et using



Malgré l'indication fournie par using ce code ne compile pas.

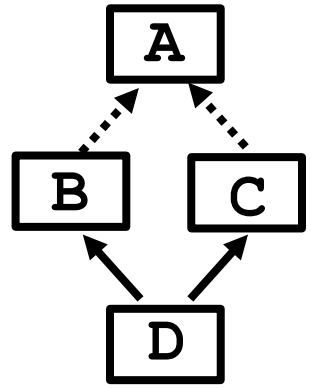
Message d'erreur: *no unique final overrider*

Justification: *C++11 Standard 10.3/2 "In a derived class, if a virtual member function of a base class subobject has more than one final overrider the program is ill-formed."*

Exemple 6

```
class A
{
public:
  virtual void f() const { cout << "A "; }
};

class B : public virtual A
{
public:
  void f() const { cout << "B "; }
};

class C : public virtual A
{
public:
  void f() const { cout << "C "; }
};

class D : public B, public C
{
public:
  void f() const { C::f(); }
};

int main()
{
  return 0;
}
```

Solution avec virtual pour méthode et virtual pour héritage



Solution: *implémenter* la méthode dans la classe où se trouve l'ambiguïté.

Points importants

l'héritage multiple est adapté pour la définition de classes de haut niveau à partir de concepts indépendants.

La phase de conception est particulièrement importante pour pouvoir étendre la hiérarchie de classe à la fois par dérivation classique (spécialisation) mais aussi en ajoutant des concepts supplémentaires.

Les fonctionalités du C++ sont loin d'être épuisées mais nous disposons déjà d'un outil puissant du point de vue de l'expression de solutions à la fois conceptuellement élaborées et performantes en pratique.