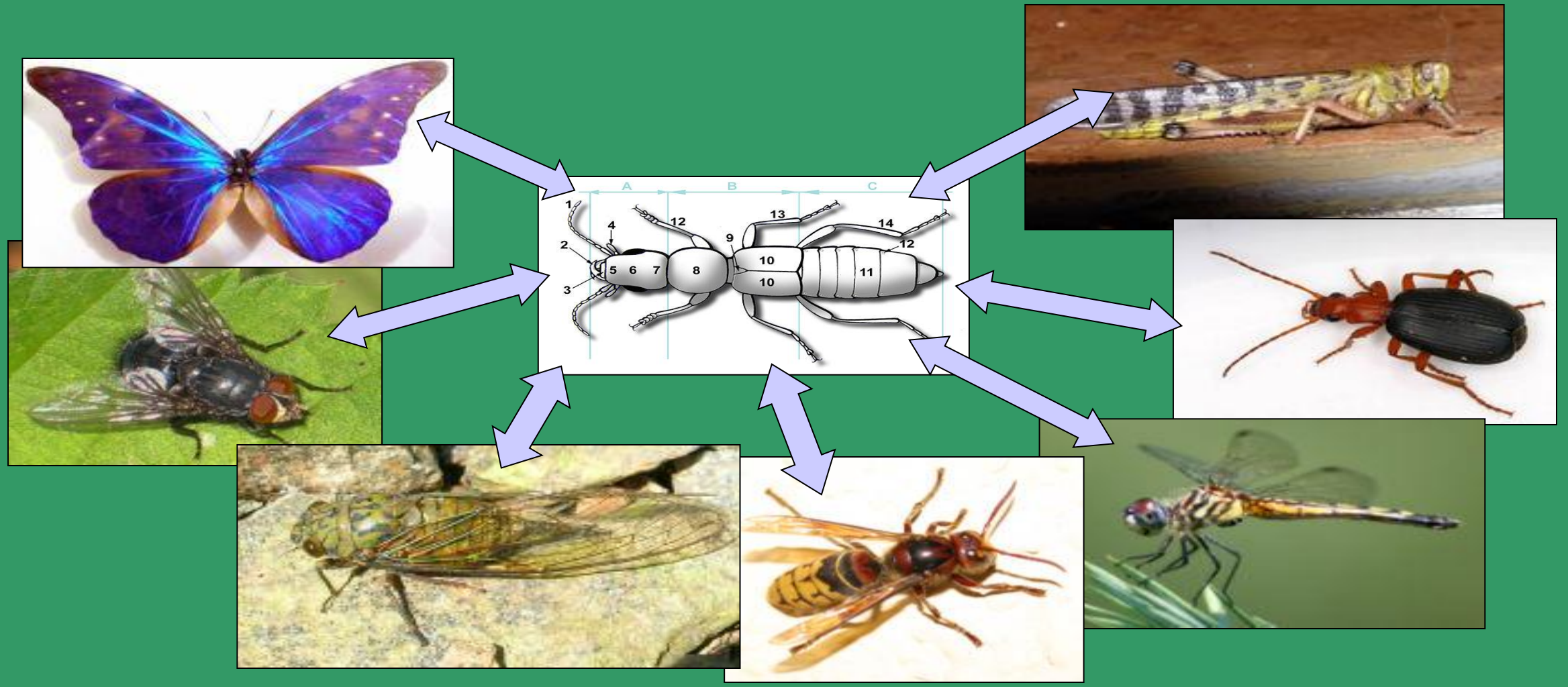
Introduction au C++
L'héritage

Pierre-Édouard Cailliau & Nicolas Gazères

Dassault Systèmes
Science & Corporate Research

Introduction entomologique

- Toutes les espèces « d'insectes » ont une organisation morphologique similaire.



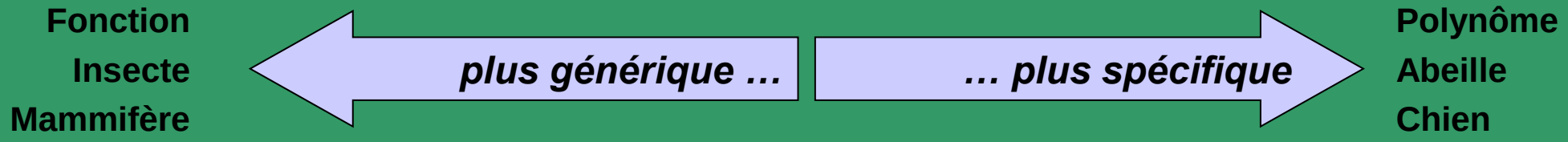
Héritage

Vision orientée objet (indépendante du langage)

- Les abstractions d'un domaine d'étude ne sont pas indépendantes les unes des autres, en général.
  - Le travail de modélisation consiste à rechercher des relations entre abstractions.
  - Il est très efficace de structurer hiérarchiquement les abstractions.
  - L'héritage est un exemple parfait de relation hiérarchique entre abstractions.

- Identifier une *relation d'héritage* entre des abstractions X et XY consiste à formaliser que XY **est « *une sorte* de »** X
  - Un polynôme « est une sorte de » fonction;
  - Une abeille est « une sorte d' » insecte.
  - Un chien est « une sorte de » mammifère.

- Dans une relation d'héritage, les abstractions sont classées du plus *générique* au plus *spécifique*.



Propriétés de la relation d'héritage

Vision orientée objet (indépendante du langage)

- **Relation entre structures**
  - La structure de X se retrouve dans XY:
    - L'abeille possède trois paires de pattes (comme tout insecte).
    - Le chien possède des mamelles, comme tout mammifère
    - Le polynôme possède un nom, comme toute fonction.
  - La structure de XY peut éventuellement augmenter celle de X:
    - Une abeille possède des cils (les insectes sont lisses, en général)
    - Un polynôme possède un degré et des coefficients (pas les fonctions générales)
- **Relation entre comportements**
  - L'abstraction XY hérite des comportements de X
    - Une chienne allaite ses chiots (comme tout mammifère)
    - Un polynôme prend une valeur pour un argument donné (comme toute fonction)
    - Une abeille pond des œufs (comme tout insecte).
  - Les comportements de XY peuvent être plus riches que ceux de X.
    - L'abeille produit du miel (pas tous les insectes).
    - Le polynôme peut se dériver (pas toutes les fonctions).

Traduire l'héritage en C++

Les classes dérivées

- Méthode
  - Les abstractions X et XY sont modélisées comme des classes.
  - La définition de la classe XY contient une spécification d'héritage.
  - La classe X est dite ***classe de base***.
  - La classe XY est dite ***classe dérivée de X***.
- Syntaxe

Fonction.h

```
class Fonction {
 …
};
```

Polynome.h

```
#include "Fonction.h"

class Polynome : public Fonction {
public:
    …;
};
```

spécificateur d'héritage *public*

Identifiant de classe de base

Identifiant de classe dérivée

L’héritage de structure en C++

Accessibilité des données-membres

- La structure de X se retrouve dans XY
  - Traduction : Toutes les données-membres *non-privés* de la classe de base sont accessibles dans la classe dérivée

- La structure de XY peut éventuellement augmenter celle de X
  - Traduction : La classe dérivée rajoute ses propres données-membres.

L'héritage de structure en C++

Exemples

Fonction.h

```
class Fonction {
public:
  const char * nom;
private:
  int taille;  // pour optimiser
};
```

Polynome.h

```
#include "Fonction.h"

class Polynome : public Fonction
{
public:
  Polynome(…);
  int degre ;

private:
  double a, b, c;
};
```

main.cpp

```
#include "Fonction.h"
#include "Polynome.h"

int main()
{
  Polynome p1(…);

  p1.degre;       // OK
  p1.a;           // Compile KO
  p1.b;           // Compile KO
  p1.c;           // Compile KO

  p1.nom;         // OK
  p1.taille;      // Compile KO
}
```

L’héritage de comportements en C++

Accessibilité des méthodes

- L’abstraction XY hérite des comportements de X
  - Traduction : Toutes les méthodes *non-privées* de la classe de base sont accessibles dans la classe dérivée.

- Les comportements de XY peuvent être plus riches que ceux de X.
  - Traduction : La classe dérivée rajoute ses propres méthodes.

L'héritage de comportements en C++

Exemples

Fonction.h

```
class Fonction {
public:
  double valueAt( double x );
  double operator()( double x );

private:
  void setSize( int sz );
};
```

Polynome.h

```
#include "Fonction.h"

class Polynome : public Fonction
{
public:
  Polynome(…);
  void Derive();
};
```

main.cpp

```
#include "Fonction.h"
#include "Polynome.h"

int main()
{
  Polynome p1(…);

  // Interface public de Polynome
  p1.Derive();          // OK


  // Interface public de Fonction
  p1.valueAt( 1.7 );    // OK
  p1( 3.0 );            // OK

  // Interface private de Fonction
  p1.setSize( 4 );      // KO
}
```

Contrôle d'accès aux membres de la classe de base

Spécificateur d'accès protected *(1)*

- Règles
  - *protected* est un niveau d'accès intermédiaire entre *public* et *private*.
  - Un symbole *protected* (donnée-membre ou méthode) peut être utilisé dans l'implémentation des méthodes de la classe ainsi que dans celle des méthodes des classes dérivées.
  - Il ne peut pas être utilisé par le « reste du code »
  - Il évite d'avoir à définir un accesseur public pour un symbole *private*.
- Exemple

Fonction.h

```
class Fonction {
protected:
  const char * nom;
};
```

Polynome.h

```
class Polynome : public Fonction {
public:
   void dumpValue( double x ) ;
};
```

Polynome.cpp

```
void Polynome::dumpValue( double x ) {
  printf( "%s(%lf)=%lf\n", nom, x, … );    // OK
}
```

Contrôle d'accès aux membres de la classe de base

Spécificateur d'accès protected *(2)*

- Cas général

X.h

```
class X {
protected:
    double a;
    int m1( char *p );
};
```

XY.h

```
class XY : public X {
    void m2();
};
```

XY.cpp

```
void XY::m2( ) {
   a;                    // OK
   m1( "hello" );        // OK
}
```

Scope d'une classe dérivée

main.cpp

```
main() {
  X  x1;
  XY y1;

  x1.a … ;              // KO !!
  x1.m1( "aaa" )… ;// KO !!
  y1.a … ;              // KO !!
  y1.m1( "aaa" )… ;// KO !!
}
```

Scope externe

Classes dérivées en C++

Le cas des méthodes spéciales

- Les méthodes spéciales ne sont pas héritées :
  - Les constructeurs
    - le(s) constructeur(s) simple(s)
    - le copy-constructeur
  - l'opérateur d'affectation
  - le destructeur
- Si ces méthodes spéciales ne sont pas présentes dans la classe dérivée :
  - Le compilateur C++ génère une **version par défaut** (règles usuelles).

Protocole de Construction

Héritage simple C++

- Le constructeur de la classe dérivée est responsable de l'initialisation de sa classe de base.
  - S'il ne l'appelle pas, un appel au constructeur par défaut est généré (s'il y en a un).
- Syntaxe
  - on met l'appel au constructeur de la *Base* dans la liste après le « : »
- Ordre d'appel
  - D'abord, le constructeur de la classe *Base*.
  - Puis les constructeurs des données-membres
    - dans l'ordre de déclaration des membres dans le header.
  - Enfin, le corps du constructeur de la classe
- Exemple

Derived.h

```
class Derived : public Base {
public:
   Derived( int i, double d );
};
```

Derived.cpp

```
Derived::Derived(int i, double d )
   : Base(i), a( i,d), b(d)
{ … }
```

On passe à la classe de base la partie utile des arguments d'entrée du constructeur de Derived.

Protocole de Destruction

Héritage simple

- L'ordre d'appel des destructeurs est l'ordre inverse de celui du constructeur
  - D'abord, le corps du destructeur de la classe
    - Le programmeur écrit *explicitement* cette partie (si besoin)
  - Puis les destructeurs des données-membres
    - Pour les données-membres qui ont un destructeur
    - dans l'ordre *inverse* de déclaration des membres dans le header.
    - cette partie est générée *implicitement* par le compilateur
  - Enfin, le destructeur de la classe de base.
    - cette partie est générée *implicitement* par le compilateur

Conclusion

Avantages liés à la relation d'héritage

- Factorisation
  - La présence d'une donnée publique ou d'un comportement public sur une classe de base, combinée à l'héritage permet de réutiliser tel quel la donnée ou le comportement sur une classe dérivée.
- Extensibilité
  - La relation d'héritage n'empêche pas de compléter la structures et les comportements de base par des structures et des comportements nouveaux, uniquement accessibles sur la classe dérivée.

Héritage multiple

- Il est parfois utile d'exprimer qu'une abstraction hérite de plusieurs abstractions de base *indépendantes*
  - Exemple:
    - Un Polynôme est une sorte de Fonction, mais aussi:
      - une sorte d' Objet Dérivable
      - une sorte d' Objet Sauvegardable
    - …
- Le mécanisme objet qui permet de modéliser ce type de relations est *l'héritage multiple*
  - Syntaxe:

Polynome.h

```
#include "Fonction.h"
#include "Derivable.h"
#include "Sauvegardable.h"

class Polynome : public Fonction,
                 public Derivable,
                 public Sauvegardable {
public:
    …;
};
```

Notion d'agrégation

Ne pas confondre agrégation et héritage

- Lorsqu'une abstraction X possède comme donnée constitutive une abstraction Y, on dit que ***X agrège Y***.
  - Ou que: Y est un composant de X

- Exemples
  - Un visage possède un nez, une bouche, deux yeux…
  - Un polynôme possède des coefficients et un degré.
  - Un insecte possède une carapace, six pattes, des antennes…

- L'agrégation
  - est une relation « *Has-A* »
  - Est souvent modélisée par des données-membres, pas comme une relation d'héritage.

Introduction au C++

Le polymorphisme

Pierre-Édouard Cailliau

École Centrale Paris / OMA

Introduction au polymorphisme

- Exemples (en langage naturel)
  - « le jeudi après-midi, les élèves font *du sport* ».
  - « chacun fait son plein de *carburant* ».
  - « chaque musicien joue de son *instrument* ».
- Au quotidien, nous adoptons un niveau de description du monde qui est le niveau *adapté*, pas forcément le plus *précis*.

- Le polymorphisme:
  - consiste à manipuler un objet, non-pas comme représentant de son type exact (ie. le plus spécifique), mais comme représentant d'un type englobant.
  - garantit que toute opération invoquée sur l'objet (vu comme objet de base) est malgré tout bien celle définie au niveau du type exact.
- Les avantages
  - économie d'expression, et
  - des algorithmes écrits au niveau d'abstraction du type englobant

Les principes du polymorphisme

- Le polymorphisme est une composante essentielle du système de typage des langages orientés objets.

- Il s'appuie sur deux principes:
  - **Le principe de *substitution***
    « en tout point du programme où un certain type est attendu, on peut lui substituer un type *dérivé*, direct ou indirect. » (≠C)

  - **Le principe de *liaison dynamique***
    « Sur un objet vu et manipulé comme un élément de la classe de base, tout appel de méthode (virtuelle) est rebranché vers l'implémentation spécifique du type exact de l'objet ».

Avantages du polymorphisme

- Le polymorphisme permet d'écrire les programmes-clients uniquement en fonction de l'interface des objets qui seront manipulés et du contrat qu'elle représente.
  - Chaque musicien joue de son instrument
- L'écriture initiale du programme nécessite plus d'efforts d'abstraction que pour un programme équivalent non-objet,
  - Mais les extensions faites par la suite au programme (sous la forme d'ajout de nouvelles classes) sont beaucoup plus faciles, pourvu que les nouvelles classes adhèrent aux mêmes interfaces et respectent leur contrat.
  - Exemple:
    - J'introduis un nouvel instrument. Comme tout instrument, il adhère à l'interface instrument, qui garantit qu'on peut en jouer. La phrase « chaque musicien joue de son instrument » reste vraie et inchangée, avec le nouvel instrument.
- L'évolution des programmes se faisant le plus souvent par ajouts de nouvelles classes, le *polymorphisme* permet donc un gain de productivité majeur dans le développement logiciel.

Les Conversions Standard

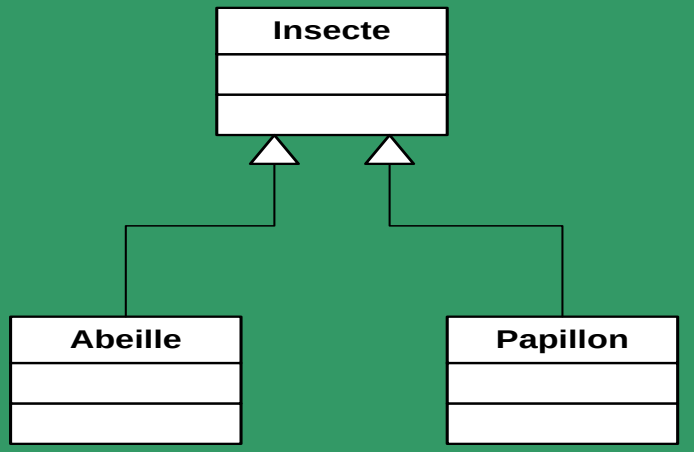
Support du principe de substitution

- Elles ont un rôle fondamental dans le polymorphisme C++:
  - *La base du principe de substitution*.
- Principe
  - On peut convertir implicitement une référence sur un objet de type *T* en une référence sur un objet d'un type de base de *T*.
  - On peut convertir implicitement un pointeur sur un objet de type *T* en un pointeur sur un objet d'un type de base de *T*.
- Exemple

```
class Abeille  : public Insecte { … };
class Papillon : public Insecte { … };

Abeille  maya;
Papillon gaston;

Insecte & insecte1 = maya;     // OK
Insecte & insecte2 = gaston;   // OK
Papillon & pap1    = gaston;   // OK
Papillon & pap2    = maya;     // KO !!
Papillon & pap3    = insecte1; // KO !!
```



Les méthodes virtuelles

Support en C++ du principe de liaison dynamique

- Syntaxe

Insecte.h

```
class Insecte {
public:
   virtual char * Couleur();
};
```

mot-clé réservé

Les méthodes virtuelles

Exemple (1)

Insecte.h

```
class Insecte {
public:
    virtual char * Couleur();
};
```

Abeille.h

```
#include "Insecte.h"

class Abeille : public Insecte {
public:
   char * Couleur();
};
```

le spécificateur *virtual* n'est plus nécessaire pour la classe dérivée

Insecte.cpp

```
char * Insecte::Couleur() {
  return "impossible à dire";
};
```

implémentation pour la classe de base

Abeille.cpp

```
char * Abeille::Couleur() {
  return "Noir et jaune";
};
```

surcharge = nouvelle implémentation pour la classe dérivée

L’appel de méthode virtuelle

Support du principe de liaison dynamique

main.cpp

```
int main() {
  Insecte i1;
  Abeille a1;
  Papillon p1;
  Insecte *pi = 0;

  pi=&i1;
  printf("%s",pi->Couleur() );

  pi=&a1;
  printf("%s",pi->Couleur() );

  pi=&p1;
  printf("%s",pi->Couleur() );

  return 0;
}
```

```
// Affichage associé:

→Impossible à dire


→Noir et jaune


→Bleu
```

Les méthodes virtuelles

Résumé

- *Une méthode virtuelle a vocation à être redéfinie par une classe dérivée*.
- Une classe dérivée peut redéfinir l'implémentation de la méthode virtuelle lorsque l'implémentation de la classe de base n'est pas adaptée.
  - On parle de « *surcharge* »
  - Sur la classe dérivée, la signature de la méthode doit être :
    - la même, ou
    - covariante (voir plus loin)

- *Règle fondamentale*
  - L'implémentation dans la classe dérivée doit être en tout point ***conforme à l'esprit du contrat*** défini pour la méthode virtuelle.

Signatures covariantes

- Les signatures covariantes sont une façon de relier:
  - le type de retour d'une méthode virtuelle dans la classe de base, et
  - le type de retour de la méthode qui surcharge cette méthode dans la classe dérivée.

Surcharge traditionnelle

```
class X;

// La méthode virtuelle
// d'origine renvoie un X*.
class A {
  public:
    virtual X* f();
};

// La méthode qui surcharge doit
// avoir le même type de retour.

class B : public A {
  public:
    X* f();
};
```

Surcharge covariante

```
class X { … };
class Y : public X { … ];

// La méthode virtuelle
// d'origine renvoie un X*.
class A {
  public:
    virtual X* f();
};

// La méthode qui surcharge renvoie
// un type covariant avec X.

class B : public A {
  public:
    Y* f();
};
```

Mécanisme des méthodes virtuelles

Exemple récapitulatif d'utilisation

Insecte.h

```
class Insecte {
public:
    virtual void vmeth1();
    virtual void vmeth2();
    virtual void vmeth3();
    void meth4() ;
};
```

Abeille.h

```
class Chat : public Animal {
public:
   void vmeth1();          // virtuelle
   void vmeth2( int );     // autre méthode !
   int  vmeth3();          // erreur de compilation
   void meth4();
};
```

main.cpp

```
Abeille maya;
Insecte * pi = &maya;      // conversion standard!

pi->vmeth1();              // Abeille::vmeth1
pi->vmeth2();              // Insecte::vmeth2
pi->vmeth2( 45 );          // erreur de compilation
pi->meth4();               // Insecte::meth4 (non-virtuelle)
```

L'appel d'une méthode virtuelle dépend du type exact de l'objet pour lequel elle est invoquée.

L'appel d'une méthode non-virtuelle dépend du type de la *référence* (resp. du *pointeur)* sur lequel elle est invoquée.

Destructeur virtuel (1)

- Contexte
  - Lorsqu'une classe dérivée possède des données-privées
    - Par exemple, un pointeur vers des données sur le Tas
  - et que la destruction d'un objet est invoquée via un « pointeur sur classe de base »,
  - alors, le destructeur doit avoir été déclaré *virtual* sur la classe de base pour que ce soit le destructeur du *type exact* de l'objet qui soit invoqué (ie. le destructeur du type *dérivé*).
  - Sinon c'est le destructeur de la classe du pointeur qui est invoqué
    - problème: il n'a aucune connaissance de l'existence de la donnée privée de sa classe dérivée → fuite-mémoire
- Recommandation
  - Le destructeur de la classe de base doit être virtuel s'il est possible qu'un delete soit invoqué sur un « *pointeur sur classe de Base* ».

Destructeur Virtuel (2)

Exemple

Container.h

```
class Container {
public:
    virtual ~Container() { }
};
```

Tableau.h

```
#include "Container.h"

class Tableau : public Container {
public:
   Tableau( int n );
   ~Tableau();

private:
   char * p;
};
```

Tableau.cpp

```
#include « Tableau.h"

Tableau::Tableau( int n ) {
    p=new char [ n ] ;
}

Tableau::~Tableau() {
    delete [ ] p;
}
```

main.cpp

```
{
   …
   Container * pc = new Tableau(10);
   …
   delete pc;    // ????
```

Appel via pointeur + destructeur virtuel → donnée-membre désallouée.

Mécanisme des méthodes virtuelles

Règle fondamentale

- Le mécanisme des méthodes virtuelles n'est déclenché que sur un pointeur ou sur une référence.
  - Avec un *pointeur*, il peut y avoir différence entre le type du pointeur et le type réel de l'objet pointé.
    - Idem pour une *référence.*
  - Par contraste, avec un *objet*, il n'y a aucun polymorphisme possible
    - Le type exact est forcément le type manipulé.

- Règles
  - Lorsque la méthode est **virtuelle**, c'est le **type exact de l'objet pointé** qui détermine la méthode choisie.
  - Lorsque la méthode n'est **pas virtuelle**, c'est le **type du pointeur** qui détermine la méthode choisie.

Méthode virtuelle *pure*

- Une méthode virtuelle dont la déclaration est suivie d'un « =0 ».

```
class Insecte {
public:
    virtual char * Couleur() = 0;
};
```

méthode virtuelle *pure*

- Conséquence
  - Lorsqu'une classe possède au moins une méthode virtuelle pure, le C++ garantit par construction que cette classe est *non-instanciable.*
  - Une telle classe est dite « *classe abstraite* » (ou une « interface »)
  - Seule une classe dérivée de la classe abstraite qui implémente toutes les méthodes virtuelles pures redevient instanciable.
- Attention:
  - Une méthode virtuelle pure peut fournir une implémentation
    - la classe reste néanmoins abstraite (donc non-instanciable)
    - l'implémentation peut servir d'implémentation par défaut pour les classes dérivées (cf. rappel de l'implémentation de base)

Interface

- Avez-vous déjà vu « un insecte » ?
  - La réponse est non
- Définition
  - Une interface est la forme la plus épurée du contrat qui lie un fournisseur de service et un client de ce service.
    - Si un insecte passe devant vous, vous savez:
      - qu'il peut voler
      - qu'il a trois paires de pattes…
  - Une interface représente:
    - une abstraction suffisamment générique pour servir de classe de base.
    - tellement générique qu'elle n'a pas de sens concret dans le monde réel.
    - un contrat que seule une spécialisation sait implémenter.
- Formalisation en C++
  - Une interface est une classe abstraite
    - de préférence sans données-membres
  - Seule une classe dérivée de l'interface est instanciable.
    - à condition de fournir une implémentation pour toutes les méthodes virtuelles pures.

Recommandations

Programmation orientée objet

- Identifiez les relations entre abstractions du domaine.
  - Héritage (« est une sorte de »)
  - Agrégation (« contient »)
- Introduisez des interfaces
  - implémentez vos algorithmes en termes de ces interfaces
- Les interfaces doivent être:
  - homogènes
    - toutes les méthodes ont un rapport entre elles.
  - minimales
    - le minimum de méthodes pour fournir le service souhaité.
- On évite en général d'avoir des données-membres sur les interfaces (mais ça peut arriver).

Recommandations

Langage C++

- Factorisez les *comportements* communs à plusieurs classes dérivées dans la classe de base
  - comme méthodes *protected*
- Mettez les *structures* communes dans la classe de base
  - comme données-membres *protected*
- N'écrivez pas de méthode spéciale si ce n'est pas nécessaire
  - ie. si l'implémentation par défaut suffit
    (et elle suffit en général s'il n'y a pas de pointeur vers le Tas)
- Si vous écrivez une classe destinée
  - (1) à être dérivée,
  - (2) par des classes qui possèderont des données-membres allouées dynamiquement (Tas), alors:

  → déclarez le destructeur virtuel
- Toute méthode n'a pas besoin d'être virtuelle
  - Elle est virtuelle si elle a vocation à être surchargée dans les classes dérivées.