Polymorphisme (1)

D'une façon générale, l'adjectif «***polymorphe***» est utilisé pour désigner:
«le caractère de ce qui revêt plusieurs formes»

En programmation, on distingue 2 types de polymorphisme:

- le **polymorphisme des fonctions,**[1]
  représente le caractère polymorphe de séquences d'instructions; essentiellement visible avec le **mécanisme de surcharge**: un même identificateur est utilisé pour désigner des séquences d'opérations différentes.

- le **polymorphisme des données,**[2]
  pour lequel on distingue encore entre:

  ⇨ le **polymorphisme *d'inclusion***:
  lorsqu'un même code peut être appliqué à des données de types différents, si tant est que ces types sont liés entre eux par une relation de sous-typage, comme c'est le cas avec les hiérarchies de classes.

  ⇨ le **polymorphisme *paramétrique***:
  lorsqu'un même code peut être appliqué à des données de n'importe quel type (on parle aussi de *généricité*), comme c'est le cas des `vector`

1. Appelé ***polymorphisme ad hoc***
2. Appelé ***polymorphisme universel***

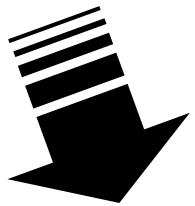
Polymorphisme d'inclusion (1)

Dans les langages objet, le polymorphisme d'inclusion est la possibilité, à chaque fois qu'une instance d'une classe Θ est «attendue» (en argument d'une fonction ou lors d'affectations), d'utiliser, *à sa place*, une instance de n'importe quelle sous-classe θ de Θ.

Exemple

```
class A {
public:
   int a;
   ...
};
class B : public A {
public:
   int b;
   ...
};
```

```
void affiche(const A obj) {
   cout << "Etat: ";
   cout << obj.a << endl;
}
int main() {
   B bb; bb.a = 4; bb.b = 3;
   A aa; aa = bb;
   affiche(aa);
   affiche(bb);
}
```



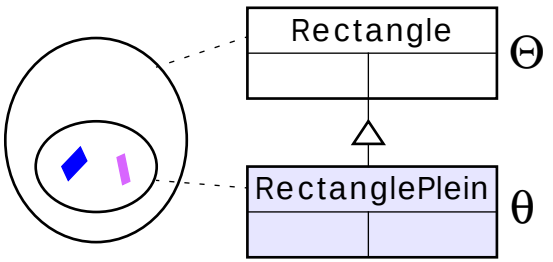
```
Etat: 4
Etat: 4
```

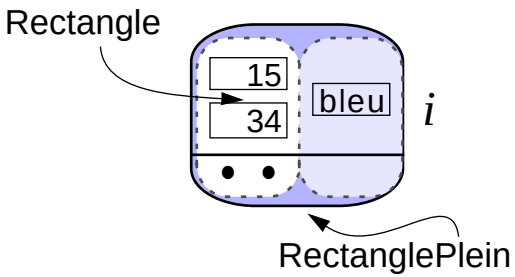
Cet exemple illustre donc le fait que les instances d'une sous-classe (ici B) sont ***substituables*** aux instances des classes de leur ascendance (ici A).

Polymorphisme d'inclusion (2)

Notons que c'est le mécanisme d'héritage qui permet de réaliser ce polymorphisme:[3]

En effet, l'héritage consistant à enrichir la description des classes en ajoutant des propriétés, **sans jamais remplacer ou supprimer celles existantes**, il permet de considérer qu'une instance *i* d'une sous-classe θ **est aussi** une instance de la classe parente Θ et, par extension, de toutes les classes de l'ascendance de θ.





RectanglePlein ⊂ Rectangle

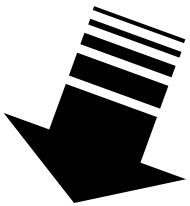
3. Comme on l'a vu, il ne s'agit que de l'une des formes possibles de polymorphisme; // c'est toutefois celle à laquelle il est le plus souvent fait référence lorsque l'on parle de polymorphisme en C++

Polymorphisme d'inclusion (3)

Cependant, le polymorphisme limité à la seule compatibilité ascendante des types en cas d'héritage n'est, à lui seul, pas très intéressant:

```
class A {
public:
   int a;
   void dump() const {
      cout << "a=" << a;
   }
};
class B : public A {
public:
   int b;
   void dump() const {
      A::dump();
      cout << ",b=" << b;
   }
};
```

```
void affiche(const A obj) {
   cout << "Etat: ";
   obj.dump();
   cout << endl;
}
int main() {
   B bb; bb.a = 4; bb.b = 3;
   A aa; aa = bb;
   affiche(aa);
   affiche(bb);
   bb.dump();
}
```



```
Etat: a=4
Etat: a=4
Etat: a=4,b=3
```

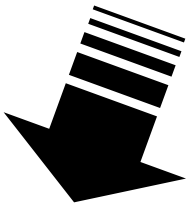
Bien que redéfinie dans la sous-classe `B`,
la méthode `dump()` invoquée depuis la fonction `affiche()` est toujours celle de la classe `A`.
Ceci est tout à fait normal, puisque `affiche()` n'utilise pas l'instance passée
en argument (`bb`), mais une copie (passage par valeur), qui elle est de type `A`.

Polymorphisme d'inclusion (4)

Une partie de la solution au problème précedent est de recourir, non pas à un passage par valeur, mais à un passage **par référence**:[4]

l'instance n'est en effet dans ce cas plus dupliquée, et c'est donc bien sur celle passée en argument que la fonction agira:

```
class A {
public:
   int a;
   void dump() const {
      cout << "a=" << a;
   }
};
class B : public A {
public:
   int b;
   void dump() const {
      A::dump();
      cout << ",b=" << b;
   }
};
```



```
Etat: a=4
Etat: a=4
Etat: a=4,b=3
```

```
void affiche(const A& obj) {
   cout << "Etat: ";
   obj.dump();
   cout << endl;
}
int main() {
   B bb; bb.a = 4; bb.b = 3;
   A aa; aa = bb;
   affiche(aa);
   affiche(bb);
   bb.dump();
}
```

4. On peut aussi utiliser la notion de pointeur (qui sera étudiée plus tard dans le cours).

Polymorphisme d'inclusion (4)

Cependant, le recours aux références ne constitue **qu'une partie de la solution**, car même dans ce cas, **c'est toujours la méthode originelle qui est invoquée**.

La raison est simple: la définition originelle de la méthode **fait aussi partie** de la sous-classe, elle est héritée comme toutes les autres propriétés.

Par conséquent, lors de la *résolution de liens* (i.e. du choix de la définition à utiliser lors de l'invocation d'une méthode), le compilateur utilise la définition de la méthode associée à la classe imposée par **le contexte statique** de l'invocation de la méthode (dons notre exemple le type de l'argument).

Résolution dynamique des liens

La solution est alors de permettre
une **résolution dynamique des liens**,

i.e. le choix, lors de l'exécution – et non plus au
moment de la compilation – des méthodes à invoquer,
en fonction de la nature réelle des instances concernées.

Méthodes virtuelles (1)

En C++, on indiquera au compilateur qu'une méthode peut fair l'objet d'une résolution dynamique des liens en utilisant, **lors du prototypage originel** de la méthode[5] (i.e. dans la classe la plus générale qui admet cette méthode), le mot-clef «**`virtual`**»:

*[***`virtual`***]* <*type de retour*> <*nom de méthode*>**(**<*arguments*>**)** *[*`const`*];*

Les méthodes ainsi définies sont appelées des ***méthodes virtuelles***

```
class A {
public:
   int a;
   virtual void dump() const {
      cout << "a = " << a;
   }
};
```

Les redéfinitions éventuelles, dans les sous-classes, de la méthode seront elles aussi implicitement considérées comme virtuelles (transitivité).

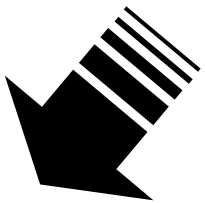
5. Remarquons bien que le programmeur n'indique pas explicitement quels appels (invocations) doivent être résolus statiquement ou dynamiquement, mais bien quelles méthodes sont susceptibles de devoir être liées dynamiquement, le compilateur se chargeant d'identifier les appels nécessitant une résolution dynamique.

Méthodes virtuelles (2)

Exemple 1:

```
class A {
public:
   int a;
   virtual void dump() const {
      cout << "a=" << a;
   }
};
class B : public A {
public:
   int b;
   void dump() const {
      A::dump();
      cout << ",b=" << b;
   }
};
```

```
void affiche(const A& obj) {
   cout << "Etat: ";
   obj.dump();
   cout << endl;
}
int main() {
   B bb; bb.a = 4; bb.b = 3;
   A aa; aa = bb;
   affiche(aa);
   affiche(bb);
   bb.dump();
}
```



```
Etat: a=4
Etat: a=4,b=3
Etat: a=4,b=3
```

Méthodes virtuelles (3)

Exemple 2:

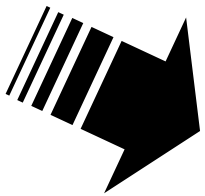
```
class Position {
public:
   int x,y;
   // ...
   virtual void dump() const {
      cout « "2D (" « x « ',';
      cout « y « ')' « endl;
   }
};

class Position3D : public Position {
public:
   int z;
   // ...
   void dump() const {
      cout « "3D (" « x « ',';
      cout « y « ',';
      cout « z « ')' « endl;
   }
};
```

```
void affiche(const Position& p)
{
   p.dump();
}

int main() {
   Position p2d(1,1);
   Position3D p3d(3,7,5);
   affiche(p2d);
   affiche(p3d);
}
```

(en supposant définis les constructeurs appropriés)



```
2D (1,1)
3D (3,7,5)
```

la référence `p` de la fonction `affiche()` est donc dans ce cas véritablement ***polymorphe***

Méthodes virtuelles (4)

Lorsque l'on **invoque une méthode virtuelle à partir d'une référence** [ou d'un pointeur] vers une instance, c'est donc la définition de la méthode associée à la classe correspondant **au type réel de l'instance** qui sera utilisée lors de l'exécution.

Attention

- Un constructeur ne peut pas être *virtuel*.
- Il ne faut pas invoquer de méthode virtuelle depuis un constructeur car dans ce cas, la résolution de liens est toujours statique.
- Il est conseillé de toujours définir les destructeurs comme étant *virtuels* .[31]

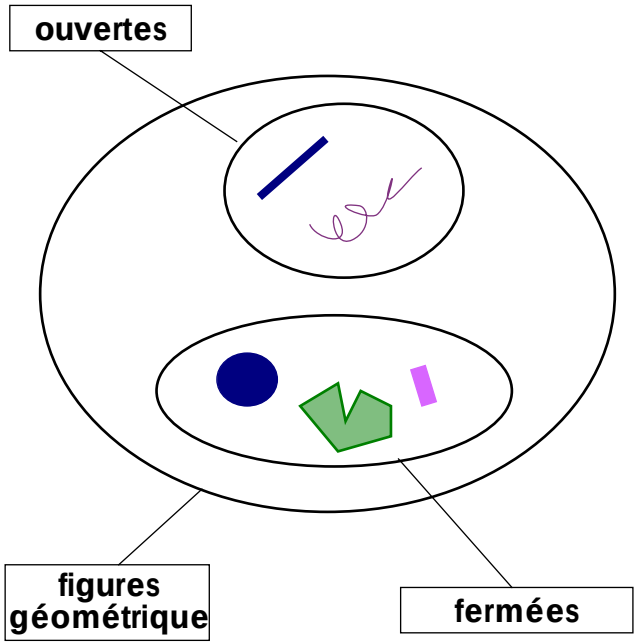
6. Certains auteurs conseillent même de définir systématiquement *virtuelles* toutes les méthodes [hormis les constructeurs], pour ne pas limiter l'extensibilité par héritage de la classe.

Méthodes (et classes) abstraites (1)

Lors de la définition d'une hierarchie de classes,
il n'est pas toujours possible de donner, au sommet de la hiérarchie,
une définition générale des toutes les méthodes, compatible avec les différentes sous-classes, ceci même si l'on sait par contre que toutes ces sous-classes vont effectivement implémenter ces méthodes.

Exemple:

Considérons que que l'ensemble des *figures géométriques* (planes) se subdivise en l'ensemble des figures *ouvertes* et l'ensemble des figures *fermées*, ce dernier étant lui-même subdivisé en l'ensemble des polygones (...), et l'ensemble des ellipses (...).



Même si l'on sait que, **pour toutes** les figures fermées, les notions de *surface* et de *périmètre* ont un sens,[7] on ne peut pas forcément définir ces notions de manière à la fois générale et précise (opératoire, i.e. directement calculable), pour l'ensemble des figures géométriques fermées.

7. Le périmètre de certaines courbes fractales pouvant éventuellement être infini.

Méthodes (et classes) abstraites (2)

Il est cependant tout à fait probable que l'on ait à écrire des fonctions ou méthodes qui devront manipuler des grandeurs comme la surface et le périmètre, mais ceci indépendamment de la nature précise des figures concernées

(on pourrait par exemple penser à une fonction permettant de déterminer la quantité de peinture nécessaire pour peindre des figures)

Du fait de leur généralité (indépendance par rapport à la nature précise de la figure fermée) ces fonctions pourront être définies comme manipulant des références de type «figures fermées», mais à l'aide de fonctions spécifiques (calcul de surface) qui, elles, devront rester sensible à la nature des figures manipulées.

Ces fonctions spécifiques devront donc être définies au niveau de la classe des figures fermées comme des méthodes virtuelles pour permettre une résolution dynamique des liens ... sans pour autant pouvoir être associées à un corps précis, puisqu'on ne dispose pas pour elles d'une définition générale et opératoire.

Dans ce cas, plutôt que d'associer à ces méthodes virtuelles une définition «bidon» qui sera de toutes les façons redéfinies dans les sous-classes, C++ offre la possibiliter de les déclarer comme des **méthodes *virtuelles pures*** (également appelées **méthodes *abstraites***), i.e. comme des méthodes virtuelles sans définition dans la classe de base.

Méthodes (et classes) abstraites (3)

La déclaration d'une méthode ***virtuelle pure***
se fait en prototypant la méthode de la manière suivante:

virtual <*type de retour*> <*nom de méthode*>**(**<*arguments*>**)** [const] **= 0**;

Exemple:

```
class FigureFermee: public FigureGeometrique {
public:
   virtual double surface() const = 0;
   virtual double perimetre() const = 0;
   // ...
};
```

Comme les méthodes déclarées virtuelles pures n'ont pas de définition au niveau de la classe dans laquelle elles sont déclarées, la classe en question est **incomplètement spécifiée**;

on appelle **classe *abstraite*** une classe qui admet au moins une méthodes *virtuelle pure*, et i**l n'est pas possible d'instancier une telle classe** [puisqu'elle n'est pas complêtement spécifiée].

Méthodes (et classes) abstraites (4)

Les sous-classes d'une classe abstraite restent elles-mêmes abstraites si elles ne fournissent pas de définition pour toutes les méthodes virtuelles pures dont elles héritent.

sous-classe entièrement spécifiée (non abstraite)

```
class Cercle:public FigureFermee {
public:
   double surface() const {
      return (PI*rayon*rayon);
   }
   double perimetre() const {
      return (2*PI*rayon);
   }
protected:
   double rayon;
};
```

sous-classe incomplètement spécifiée (abstraite)

```
class Polygone:public FigureFermee {
public:
   double perimetre() const {
      double p(0.0);
      for(int i(0);i<cotes.size();++i)
         p += cotes[i];
      return p;
   }
protected:
   vector<double> cotes;
};
```

Dans cet exemple, la classe `Cercle` n'est pas abstraite car elle donne une définition à toutes les méthodes virtuelles pures dont elle a hérité; elle peut donc être instanciée, ce qui n'est pas le cas de la classe `Polygone`, qui ne donne pas de définition à la méthode `surface()`; tout comme sa classe parente, `FigureFermee`, la classe `Polygone` est donc abstraite.

Spécialisation et partitionnement (1)

Le fait que les classes abstraites ne soient pas instanciables, et que de surcroît il n'est pas permis de rendre abstraite une sous-classe d'une classe qui n'est elle-même pas abstraite (i.e. il n'est pas possible de déclarer des méthodes virtuelles pures dans une sous-classe d'une classe non abstraite)[8] permet de donner une interprétation plus précise aux deux grands types d'utilisation de la notion d'héritage dans le contexte de la modélisation objet:

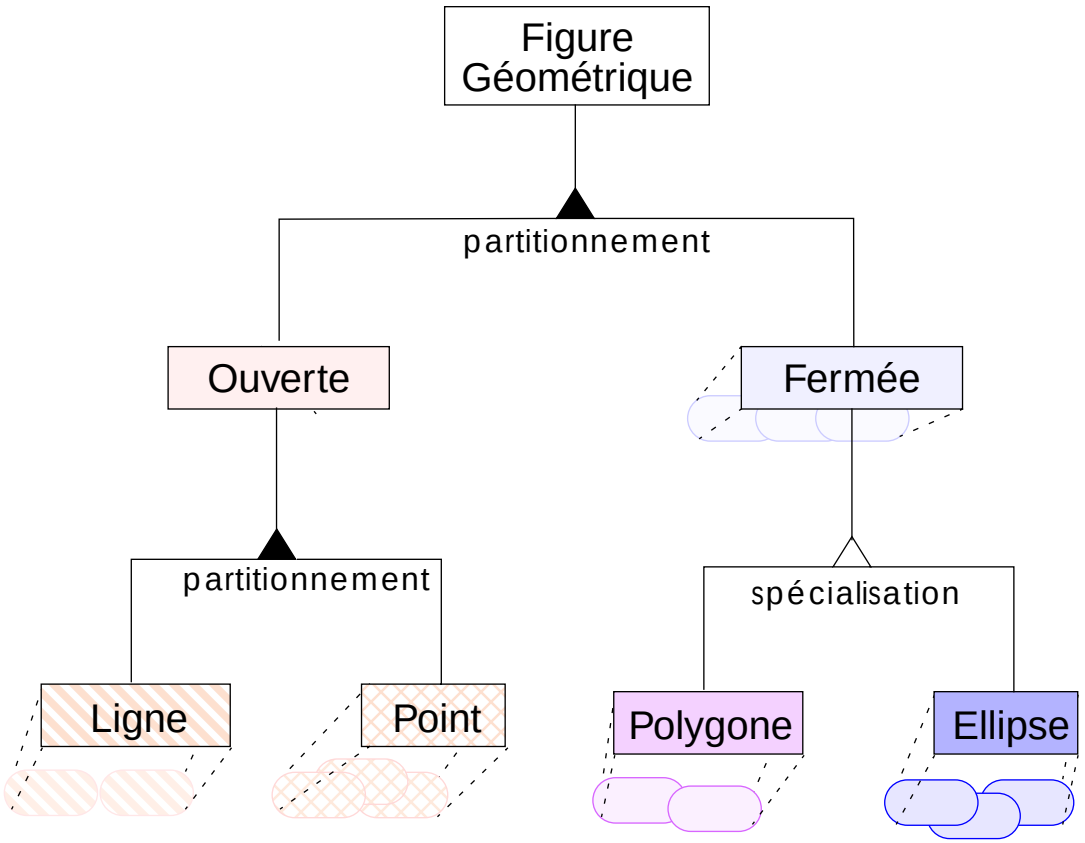
- le ***partitionnement*** = héritage à partir d'une classe abstraite:
  la classe est partitionnée en l'ensemble de ses sous-classes directes, et l'union (ensembliste) des instances de ces sous-classes représente l'extension de la classe. Une autre manière de voir le partitionnement est de considérer que la (sur-)classe est une généralisation de ses sous-classes, qui en factorise les/leur propriétés communes.

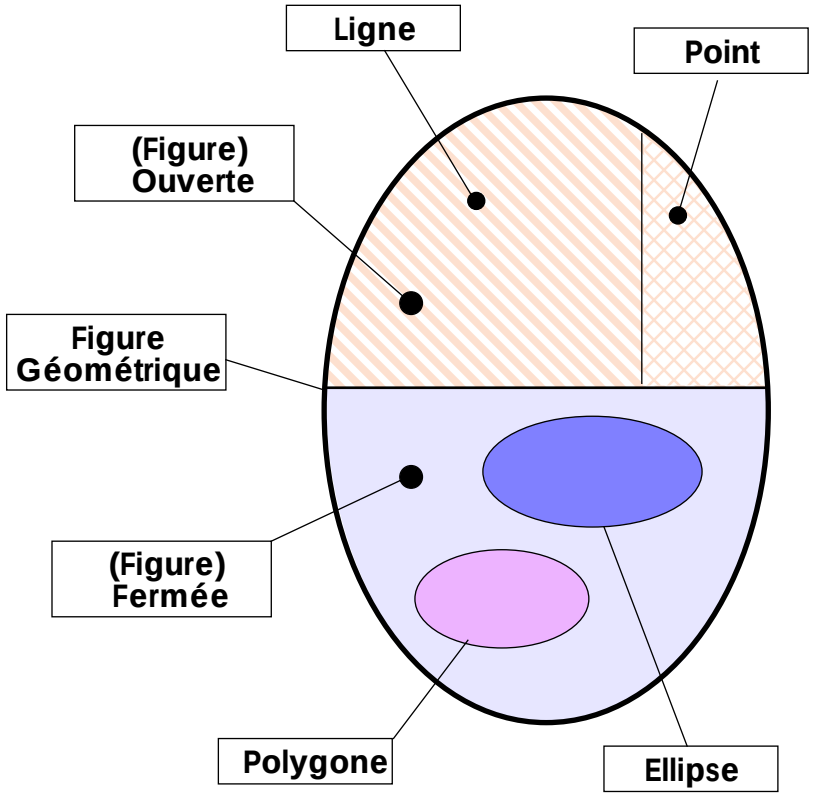
- la ***spécialisation*** = héritage à partir d'une classe non abstraite:
  l'extension de la classe est constituée de l'ensemble des instances directes de la classe augmenté de l'ensemble des instances de ses sous-classes. En d'autre terme, les instances de la classe et celles de sa descendance co-existent indépendemment les unes des autres.

8. En d'autres termes, toute l'ascendance d'une classe abstraite doit être (est) également abstraite.

Spécialisation et partitionnement (2)

Ainsi, une version plus précise de l'organisation (hiérarchique et ensembliste) des figures géométriques présentées précedemment pourrait être:





Approche modulaire et réutilisabilité

Les éléments structurels généraux du langage,
tels que les **fonctions** et les **classes**, permettent la conception de programmes dans le cadre d'une **analyse descendante**:

tant les approches *fonctionnelles* que les approches *modulaires* tendent à **décomposer la tâche à résoudre en sous-tâches** implémentées sous la forme de *fontions élémentaires* (approche fonctionnelle) ou de *modules* plus ou moins *génériques* ou *autonomes* (approche modulaire), mais, dans les deux cas, en produisant **des éléments réutilisables**.

La conception d'un programme doit alors
[entre autres] tenir compte de deux aspects importants:

- la **réutilisation** d'éléments (classes ou fonctions) existants;

- l'éventuelle **réutilisabilité** des éléments nouvellement créés.[1]

Les éléments existants peuvent se trouver stockés dans des **bibliothèques** (librairies) **de classes** ou **de fonctions**.

1. Remarquons cependant que la conception de librairies ou bibliothèques (telle la Librairie Standard de C++) est une tâche très particulière et délicate.

Exemple de conception modulaire (1)

A titre d'exemple de conception modulaire,
nous présenterons ici une version simplifiée de la classe `TableauNoir`,
classe qui sera utilisée dans le cadre de quelques exercices.

En effet, cette classe a été conçue:

- en utilisant un **ensemble de classes et fonctions pré-existantes**
  (une bibliothèque graphique spécifique – *QT* – et naturellement *STL*,
  la librairie standard de C++)

  ☞ réutilisation de l'existant

- **pour être réutilisée** par d'autres programmes
  (les programmes de certains des exercices)

  ☞ réutilisabilité

Exemple de conception modulaire (2)

Les classes réutilisées par `TableauNoir` sont issues des classes de la librairie graphique *QT* [trolltech], ainsi que de la librairie standard *STL*:

Par exemple:

- **`QWidget`**: une classe permettant de manipuler des fenêtre sur l'écran;
- **`QColor`**: une classe modélisant les couleurs;
- **`QPalette`**: une classe permettant de gérer des palettes de couleurs (de `QColor`);
- **`QPen`**: une classe représentant des crayons;
- **`QPainter`**: une classe définissant des objets permettant de dessiner (contexte d'affichage, et méthodes pour l'initialisation de crayons (`setPen`) et le dessin de points ou de lignes (`drawPoint`, `drawLine`));
- **`vector`**: l'incontournable classe modélisant des collections d'objets.

Exemple de conception modulaire (3)

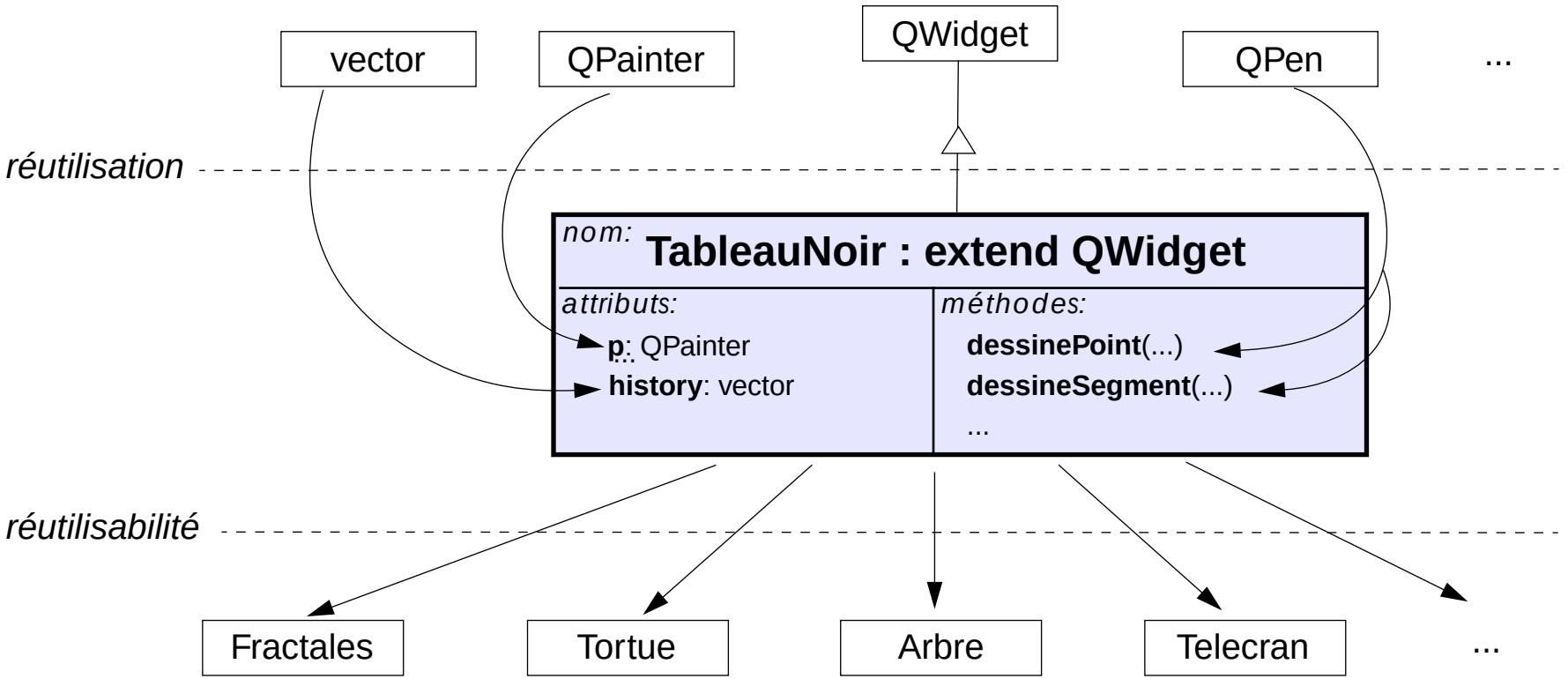
Le tableau noir que l'on cherche à définir est en fait une fenêtre de fond noir, dans laquelle il doit être possible de dessiner.

Il est implémenté par une classe `TableauNoir`, définie comme sous-classe de `QWidget` (fenêtre), et munie [entre autres] d'un attribut de type `QPainter`, permettant de réaliser les actions de dessin:

```
class TableauNoir : public QWidget
{
public:
   // Constructeurs et destructeurs:
   ...
   // prototypage des méthodes:
   void dessinePoint(float x, float y, int couleur);
   void dessineSegment( float x1, float y1,
                        float x2, float y2, int couleur);

protected:
   // attributs protégés:
   QPainter p;
   ...
};
```

Exemple de conception modulaire (4)



La classe `TableauNoir` **réutilise** l'existant (librairie graphique),
et «fournit du **réutilisable**», dans les programmes des exercices de dessin des séries.

Compilation séparée (1)

- La partie **déclarative** (déclaration des classes avec prototypes des méthodes, déclaration de variables ou constantes globales, prototypage de fonctions globales, ...) constitue la partie visible du module que l'on écrit, et qui va permettre sa réutilisation (nécessaire à la compilation)

- La partie **définition** (corps des méthodes, corps des fonctions globales, valuation des constantes, ...) est la partie d'implémentation, et n'est pas directement nécessaire pour l'utilisation du module (pas nécessaire pour la compilation)

Il est de ce fait souvent utile de séparer ces parties en deux fichiers distinct:

⇨ les fichiers de déclaration (fichier *headers*, par convention sans extension dans le cas d'une librairie C++, ou avec `.h` – parfois `.hcc` ou `.hpp`). Ce sont ces fichiers que l'on inclut en début de programme au moyen de la directive `#include`

⇨ les fichiers de définitions (fichiers sources, avec une extension `.cc` ou `.cpp`)

Compilation séparée (2)

tableau_noir.h

```
#include <vector>
#include "qwidget.h"
#include ...

class TableauNoir
   : public QWidget
{
public:
   ...
   void dessinePoint(...);
   void dessineSegment(...);
   ...
protected:
   QPainter p;
   ...
};
```

tableau_noir.cc

```
#include "tableau_noir.h"

TableauNoir::TableauNoir()
:  QWidget(...)
   ...
{ ... }

void TableauNoir::dessinePoint(...)
{
   ...
}

void TableauNoir::dessineSegment(..)
{
   ...
}
```

Compilation séparée (3)

La séparation des parties déclaration et définition en deux fichiers distincts permet une compilation séparée de ce module:

Phase 1:

production du fichier binaire (appelé fichier objet) correspondant à la compilation des fichiers sources contenant les parties définitions des modules de la librairie (ou du module, comme dans le cas de `TableauNoir`).

Le résultat de cette phase est l'assemblage en langage machine de toutes les fonctions et méthodes définies dans le module compilé.

Phase 2:

production du fichier exécutable final, à partir des fichiers objets des modules utilisés et du fichier source principal (contenant le `main()`).[2]

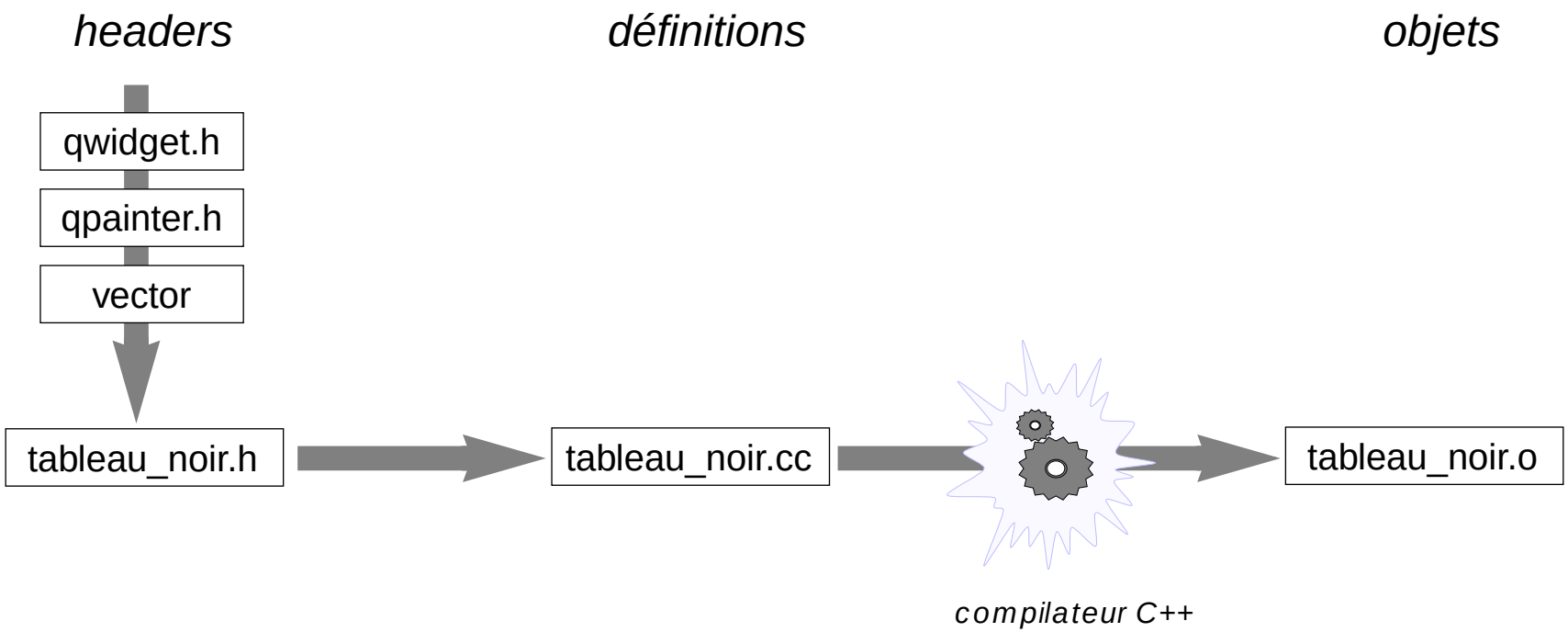
Lors de cette phase, un programme spécial, appelé éditeur de liens, extrait des fichiers objets les éléments effectivement utilisés par le programme principal, les place dans le fichier exécutable et en détermine ainsi les adresses.

2. Remarquons que le fichier source principal peut lui aussi être compilé séparément, en «Phase 1». La phase 2 consistera alors à simplement créer l'exécutable à partir de l'ensemble des fichiers objets.

Compilation séparée: phase 1

Pour créer un **fichier objet** (identifié par une extension `.o`)
on utilise une commande de compilation du type:

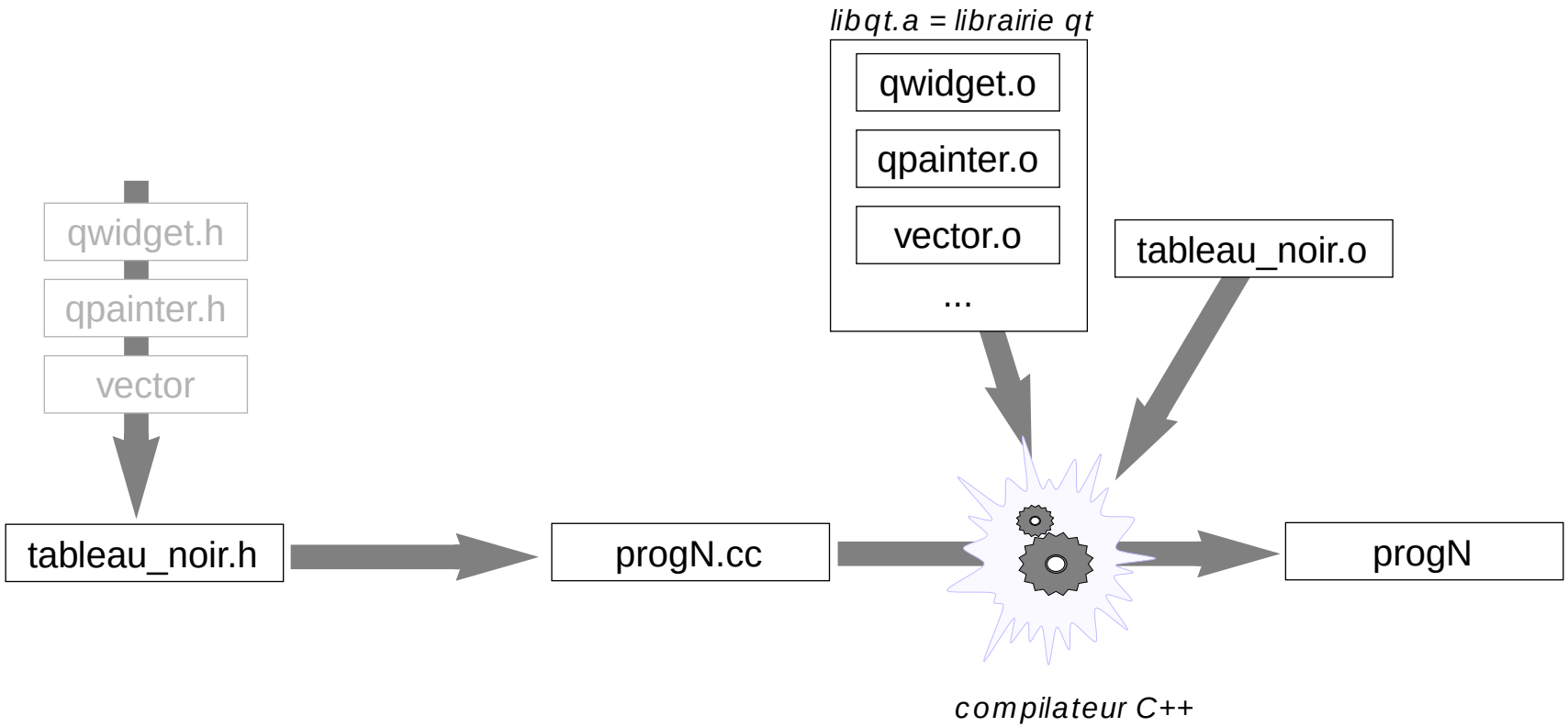
```
c++ -c tableau_noir.cc -o tableau_noir.o
```



Compilation séparée: phase 2

Lors de la seconde phase, le fichier **exécutable** est produit par une compilation qui intègre les fichiers objets nécessaires:

```
c++ progN.cc -o progN tableau_noir.o -lqt
```



Compilation séparée: synthèse