Classes, héritage, polymorphisme en C++

Ce document est un résumé du cours Programmation C++ donné au département Info de l'IUT Bordeaux 1. Il est destiné à des étudiants de l'IUT. Il ne s'agit en aucun cas d'un document de référence exhaustif. Ce document donne cependant une bonne idée des concepts de base.

Prérequis : toutes les structures de contrôles du C++ sont présumées connues, de même que la définition et l'appel des fonctions.

1 Classes et programmation orientée objet

1.1 Introduction

La programmation impérative classique sépare données et traitements sur ces données. Etant donné un type de donnée, on définit seulement quelles vont être les valeurs associées. Les différentes opérations que l'on pourra réaliser avec ce type seront définies séparément, sous forme de fonctions. Plusieurs problèmes apparaissent, entre autres :

- les valeurs ne sont pas *encapsulées*, c'est-à-dire que l'on peut partout lire ou modifier ces valeurs.
- les valeurs peuvent ne pas être correctement initialisées. La variable risque d'avoir des valeurs inconsistantes.
- les traitements étant séparés, il est difficile de connaître exactement quelles sont toutes les opérations possibles sur ce type. Il y a donc risque de ré-écriture de fonctions existantes ou de définition de traitements qui ne correspondent pas à au champ sémantique du type.

Une *classe* est un type de données dont le rôle est de rassembler sous un même nom à la fois données et traitements. Les données attachées à une classe sont appelées *attributs* (ou *données membres*). Les traitements attachés à une classe sont appelés *méthodes* (ou *fonctions membres*, *opérations*).

Une variable dont le type est une classe est appelée une *instance* de cette classe ou un *objet*.

Les méthodes sont définies de façon générique pour toutes les instances possibles d'une classe mais, à l'exécution, une méthode est reliée à une seule instance de sa classe. Ainsi, appeler une méthode n'a pas de sens en soi. On appelle la méthode sur une instance de la classe. On précisera donc toujours l'objet sur lequel la méthode s'exécute. Au sein de la méthode, on peut utiliser les noms des attributs pour accéder directement aux attributs de l'objet sur lequel la méthode a été appelée.

Le C++ permet la *programmation orientée objet*, c'est-à-dire qu'il offre un mécanisme de classe rassemblant données et traitements. Le C++ permet aussi de mélanger programmation objet et programmation impérative classique. Ainsi, on parle juste de "orientée" objet.

Le paradigme de programmation orientée objet implique une méthode différente pour concevoir et développer des applications. Les langages de modélisation objet les plus courants sont OMT et UML. Nous ne nous intéresserons pas à cet aspect de la programmation orientée objet mais plutôt aux moyens de réaliser une conception objet en C++. Sachez seulement que le paradigme objet induit notamment :

- Un découpage structurel de l'application sous forme d'objets aux rôles clairement identifiés. Les objets peuvent être reliés par de simples *associations*, par des relations d'*agrégation* (un objet est composé d'autre objets), par des relations d'*héritage* (un objet est une spécialisation d'un un autre objet).
- Un découpage des traitements sous forme de méthodes opérant sur les attributs de l'objet concerné. Lorsque deux objets sont nécessaires pour un traitement, ils dialoguent en s'envoyant des messages (*message-passing*) : schématiquement, un objet demande un service à un autre objet en appelant une méthode de cet objet avec des paramètres ; le message envoyé à l'autre objet est donc le nom de la méthode et les arguments d'appels nécessaires à son exécution.

– La possibilité de définir de façon transparente des comportements spécialisés pour des objets qui sont reliés par relation d'héritage. On parle de *polymorphisme*. En particulier, un même *message* envoyé par un objet vers plusieurs objets pourra être interprété différemment par chaque objet recepteur (suivant son type au moment de l'instanciation).

1.2 Classes en C++

En C++, on sépare l'*interface* d'une classe et son *corps*. L'interface rassemble sous le nom de la classe ses attributs et les prototypes de ses méthodes. On place en général l'interface dans un fichier entête `.h`, `.hpp`, `.H`, ou `.h++` (*nom_classe*`.h`). Le corps de la classe contient le corps des méthodes définies dans l'interface de la classe. Il est placé dans un fichier source `.cc`, `.c++`, `.C`, `.cxx`, ou `.cpp` (e.g., *nom_classe*`.cxx`).

L'exemple ci-dessous montre une classe `Personne` :

```
// Personne.h

class Personne {
private:
  string my_prenom;
  string my_nom;
public:
  Personne( string prenom, string nom );
  void quiSuisJe();
};

// Personne.cxx

// Constructeur
Personne::Personne( string prenom, string nom )
{
  my_prenom = prenom;
  my_nom = nom;
}

void Personne::quiSuisJe()
{
  cout << "Je suis " << my_prenom << " " << my_nom << endl;
}
```

Diagramme de classe UML

Personne
- string my_nom - string my_prenom
+ Personne(string prenom, string nom) + void quiSuisJe()

Exercice 1 : Définir le prototype et le corps d'une classe `Entier`, qui peut être construite à partir d'un `int`.

1.3 Instance de classe ou objet

Un *objet* est une instance d'une classe (i.e., une variable dont le type est une classe). Un objet est donc la réalisation effective d'une classe. Un objet occupe de l'espace en mémoire. Il peut être alloué :

statiquement : comme pour les variables de type de base, on écrira le nom de la classe (i.e. le type) suivi du nom que l'on veut donner à l'objet (i.e. le nom de la variable), éventuellement suivi par les arguments d'appel donnés à un constructeur de la classe. Ex :

```
Personne jpp( "Jean-Pierre", "Papin");
```

Comme toute variable déclarée statiquement, la durée de vie d'un objet est limitée au contexte dans lequel il est déclaré, en général le corps d'une fonction ou méthode. On peut aussi instancier plusieurs objets de même classe dans un tableau alloué statiquement :

```
Personne groupe[ 5 ]; // groupe est un tableau de cinq objets 'Personne'.
```

dynamiquement : cela est effectué par l'intermédiaire des opérateurs `new` et `delete`. Un pointeur sur la zone mémoire du tas où l'objet a été alloué est retourné. L'objet déclaré dynamiquement est persistant. Lorsqu'on n'en a plus besoin, on le désalloue en donnant le pointeur sur cette zone allouée à l'opérateur `delete` :

```
// Le constructeur a 2 parametres de l'objet est appele.
Personne* ptr_pers = new Personne( "Jean-Pierre", "Papin");
ptr_pers->quiSuisJe(); // Affiche 'Je suis Jean-Pierre Papin'.
delete ptr_pers;       // L'objet pointe est desalloue.
                       // NB: Le destructeur de l'objet est appele.
```

1.4 Etat d'un objet

L'*état* d'un objet est la valeur de ses attributs. Deux objets de même type sont dans le même état si leurs valeurs coïncident. Ces objets sont alors des *clones*.

1.5 Visibilité des membres d'une classe

La *visibilité* des membres d'une classe (attributs et méthodes) est définie dans l'interface de la classe. Trois mots-clés (`public`, `private`, `protected`) permettent de préciser l'accès aux membres qui les suivent dans la définition de l'interface :

public : (+ en UML) autorise l'accès pour tous. Sur l'exemple `Personne` précédent, le constructeur ainsi que la méthode `quiSuisJe` peuvent être utilisés partout sur une instance de `Personne`.

private : (− en UML) restreint l'accès aux seuls corps des méthodes de cette classe. Sur l'exemple précédent, les attributs privés `my_prenom` et `my_nom` ne sont accessibles sur une instance de `Personne` que dans les corps des méthodes de la classe `Personne`. Ainsi, la méthode `quiSuisJe` a le droit d'accès sur ses attributs `my_prenom` et `my_nom`. Elle aurait aussi l'accès aux attributs `my_prenom` et `my_nom` d'une autre instance de `Personne`.

protected : (# en UML) comme `private` sauf que l'accès est aussi autorisé aux corps des méthodes des classes qui héritent (directement ou indirectement) de cette classe. Voir la section Héritage plus loin.

1.6 Accès aux attributs et méthodes d'un objet

Etant donné une instance d'un objet, on accède à ses attributs et à ses méthodes grâce à la notation pointée ".", dans la limite de visibilité définie ci-dessus. Pour les pointeurs, cela se fait au travers de la notation fléchée "`->`". Par exemple,

```
Personne jpp( "Jean-Pierre", "Papin");
jpp.quiSuisJe();       // Affiche 'Je suis Jean-Pierre Papin'.
Personne* ptr_pers = new Personne( "Zinedine", "Zidane");
ptr_pers->quiSuisJe(); // Affiche 'Je suis Zinedine Zidane'.
delete ptr_pers;       // L'objet pointe est desalloue.
```

1.7 Constructeurs et destructeur

Les constructeurs et le destructeur sont des méthodes particulières des classes.

Les constructeurs permettent de définir un ou des comportements particuliers lors de l'instanciation d'une classe. Ils permettent notamment d'initialiser correctement un nouvel objet. Les constructeurs portent tous le nom de la classe (e.g., méthodes `Personne( ...)` pour la classe `Personne`). Deux constructeurs ont un rôle particulier :

constructeur par défaut ou constructeur sans paramètre. Il est appelé lors de l'instanciation d'un objet sans arguments d'appel. Il est aussi appelé lors de l'instanciation d'un tableau d'objets sur chacune des cases du tableau. Par exemple,

```
Personne inconnu; // appel du constructeur par defaut de 'Personne'
Personne* ptr_p = new Personne; // idem
Personne groupe[ 5 ]; // pour chaque 'Personne' du tableau 'groupe', appel
                      // du constructeur par defaut.
Personne* ptr_p2 = new Personne[ 3 ];
                      // pour chaque 'Personne' du tableau alloue dynamique-
```

```
                        // ment, appel du constructeur par defaut.
  // En tout, 1+1+5+3 = 10 appels du constructeur par defaut de 'Personne'
  // et creation de 10 instances de 'Personne'.
```

Pour une classe `Toto`, la syntaxe du constructeur par défaut est :

Syntaxe : fichier `Toto.h`

```
class Toto {
  ...
  Toto();
  ...
};
```

fichier `Toto.cxx`

```
#include "Toto.h"
...
Toto::Toto()
{ ... }
```

constructeur par copie . Il est appelé lors de l'instanciation d'un objet avec en argument d'appel un objet du même type. Il est aussi appelé lors du *passage par valeur* d'un objet en paramètre, ainsi que lors du retour d'une fonction ou méthode qui retourne un objet de ce type. Le rôle d'un constructeur par copie est de permettre l'instanciation d'un nouvel objet dans *le même état* qu'un objet existant (ou *clone*).

Syntaxe : fichier `Toto.h`

```
class Toto {
  ...
  Toto( Toto & autre );
  ...
};
```

fichier `Toto.cxx`

```
#include "Toto.h"
...
Toto::Toto( Toto & autre )
{ ... }
```

Le *destructeur* est une méthode particulière qui est définie implicitement pour toutes les classes. Son nom est de la forme $\sim$*nom_classe*. Il est appelé à la destruction/désallocation de l'objet. Par défaut, il ne fait rien. On peut lui donner un comportement spécifique. Il est indispensable lorsque l'on a besoin de faire de l'allocation dynamique.

1.8 Opérateurs, surcharge

1.8.1 Opérateurs

Il est possible de définir des opérations liées à une classe au moyen de la *surcharge* d'opérateurs existants. Si on veut définir une classe `Vecteur` dotée d'une opération "+", on peut surcharger l'opérateur "+" pour la classe `Vecteur`, en écrivant :

Syntaxe :

fichier `Vecteur.h`

```
class Vecteur {
  ...
  // prend un autre vecteur en parametre,
  // retourne un vecteur.
  Vecteur operator+( Vecteur & autre );
  ...
};
```

fichier `Vecteur.cxx`

```
#include "Vecteur.h"
...
Vecteur Vecteur::operator+( Vecteur & autre )
{ ... }
```

Il est possible de surcharger la plupart des opérateurs existants : `+ - / * == != < <= >= > ! « » ++ - += -= *= /= «= »= ...`

Exercice 2 : Reprendre l'exemple de la classe `Entier` et surcharger les opérateurs classiques (opérations, comparaisons).

1.8.2 Opérateur d'affectation

L'*opérateur d'affectation* (ou `operator=`) permet de mettre un objet *existant* (ie déjà instancié) dans le même état qu'un autre objet du même type. Cet opérateur est défini implicitement par le compilateur pour toutes les classes et structures (comme le constructeur par défaut ou le constructeur par copie). Pour une classe `X`, il est appelé à chaque fois que l'on écrit une affectation avec un objet de la classe `X` dans la partie gauche de l'affectation. Ainsi,

```
X obj; // appel du constructeur par defaut de la classe X
X obj2( "chaine qcq" ); // appel du constructeur de X prenant une chaine.
obj = obj2; // appel de l'operateur d'affectation de X.
// Attention, les trois exemples suivants ne sont pas des affectations
// mais des constructions par copie.
X obj3 = obj2; // appel du constructeur par copie de X.
X obj3( obj2 ); // equivalent
X obj3 = X( obj2 ); // equivalent
```

Son prototype est de la forme : `X & operator=( const X & autre );`

Par exemple, pour une classe `Vecteur`, cela donne :

fichier `Vecteur.h`

```
class Vecteur {
  ...
  Vecteur& operator=( const Vecteur & autre );
  ...
private:
  float my_dx;
  float my_dy;
};
```

fichier `Vecteur.cxx`

```
#include "Vecteur.h"
...
// prend un autre vecteur en parametre,
// retourne soi-meme.
Vecteur&
Vecteur::operator=( const Vecteur & autre );
{ // On fait l'affectation seulement si
  // autre est un objet different de
  // soi-meme (this).
  if ( this != &autre )
    {
      my_dx = autre.my_dx;
      my_dy = autre.my_dy;
    }
  return *this;
}
```

Un fichier source quelconque

```
Vecteur u( 10, 0 );
Vecteur v( 5, 0 );
u = v; // appel de l'operateur d'affectation => u vaut (5,0).
```

1.9 Objets passés en paramètres ou retournés

Comme toute variable, un objet peut être utilisé comme argument d'appel à une fonction ou méthode. Il existe deux types principaux de passages de paramètres :

Passage par valeur : Il est défini dans le prototype d'une fonction ou méthode avec un nom de classe suivi d'un nom de paramètre. L'objet donné en argument d'appel doit être du même type que le paramètre formel. A l'exécution de la fonction ou méthode, le paramètre formel est un *autre objet* dans le même état que l'objet donné en argument d'appel (un clone), grâce à l'appel automatique du constructeur de copie. Comme ce clone est instancié sur la pile d'exécution du processus, l'objet/paramètre formel est automatiquement détruit à la sortie de la fonction ou méthode.

Passage par référence : Il est défini dans le prototype d'une fonction ou méthode avec un nom de classe suivi du symbole "`&`" suivi d'un nom de paramètre. L'objet donné en argument d'appel doit être du même type que le paramètre formel. A l'exécution de la fonction ou méthode, le paramètre formel est le *même objet* que l'argument d'appel. Le nom du paramètre est donc un synonyme pour un objet existant. L'objet/paramètre n'est pas détruit à la sortie de la fonction ou méthode car il n'a pas été instancié dans ce contexte.

Le *passage par pointeur* est un type particulier de passage par valeur, où la valeur copiée de l'argument d'appel vers le paramètre est une adresse en mémoire. Il faut donc le voir comme un passage par valeur de type particulier.

Le *passage par référence constante* est un type particulier de passage par référence, où le paramètre formel est un synonyme d'un objet existant qui n'autorise qu'un accès en lecture à l'objet/argument d'appel.

Un objet peut être *retourné* à la fin d'une fonction ou méthode, si celle-ci définit dans son prototype une classe comme valeur de retour. A ce moment-là, c'est le constructeur par copie qui est appelé et la valeur de retour est un clone de l'objet fabriqué dans la fonction ou méthode et retourné à la fin de celle-ci.

Exercice 3 : Sur l'exemple suivant, combien d'instances de `Vecteur` sont créées ? Précisez pour chaque instance l'endroit où elle est créée et l'endroit où elle est détruite.

```
bool estColineaire( Vecteur u, const Vecteur & v )
{
  return ( u.x() * v.y() - u.y() * v.x() ) == 0;
}

Vecteur orthogonal( const Vecteur & u )
{
  Vecteur n( -u.y(), u.x() );
  return n;
}

void main()
{
  Vecteur a( 3, 2 );
  Vecteur b;
  b = a.orthogonal();
  if ( estColineaire( a, b ) )
    cerr << "Probleme !" << endl;
}
```

1.10 Méthodes constantes

Une méthode dont le prototype est terminé par `const` est appelée *méthode constante*. Dans le corps d'une méthode constante, il est impossible de modifier les données membres de l'objet, de même qu'il est impossible d'appeler des méthodes non-constantes de cet objet. En gros, un objet est en accès lecture seulement dans ses méthodes constantes.

Intérêt. Vous pouvez appeler des méthodes constantes attachées à des références constantes. Vous pouvez vérifier qu'il est impossible d'appeler une méthode non-constante sur une référence constante. Le compilateur refuse de compiler. C'est donc une protection supplémentaire pour éviter les effets de bord.

Exemples : La méthode `quiSuisJe` vue plus haut est une bonne candidate comme méthode constante. En revanche, aucun constructeur, destructeur ou opérateur d'affectation n'est une méthode constante.

Exercice 4 : Reprendre l'exemple de la classe `Entier` et désigner parmi les opérateurs surchargés les méthodes constantes.

1.11 Inclusions réciproques

Il est interdit de faire de l'inclusion réciproque dans des modules C++. Par exemple, si votre classe `Point` a besoin de la classe `Vecteur` et que votre classe `Vecteur` a besoin de la classe `Point`, alors vous ne pourrez pas écrire :

fichier `Point.h`

```
#include "Vecteur.h"
...
class Point {
...
   Vecteur toVecteur( Point p );
...
};
```

fichier `Vecteur.h`

```
#include "Point.h"
...
class Vecteur {
...
   Point toPoint();
...
};
```

Si vous avez deux classes qui ont besoin de se connaître l'une l'autre dans l'interface, alors il faut que l'une des classes n'utilise l'autre qu'au travers de *pointeurs* ou *références* dans l'*interface de la classe*. Dans ce cas-là, on n'incluera pas l'interface de l'autre classe, mais on déclarera juste que cette autre classe existe. Ci-dessous, un `Vecteur` n'utilise les `Point`s que sous forme de pointeurs dans son interface :

fichier `Point.h`

```
#include "Vecteur.h"
...
class Point {
...
   Vecteur toVecteur( Point p );
...
};
```

fichier `Vecteur.h`

```
// La classe 'Point' existe quelque part.
class Point;
...
class Vecteur {
...
  // Retourne un pointeur sur un 'Point'.
  // (certainement alloue dyn.)
  Point* toPoint();
...
};
```

fichier `Vecteur.cc`

```
#include "Vecteur.h"
// On inclue l'interface de 'Point' seulement dans le corps
// de la classe 'Vecteur'.
#include "Point.h"
...
Point* Vecteur::toPoint()
{
  // e.g., construit le Point au bout du vecteur place
  // a l'origine.
  return new Point( my_dx, my_dy );
}
```

De manière générale, si vous manipulez une classe `X` dans une interface au travers de références ou de pointeurs, il est bon de déclarer la classe dans l'interface (par un `classe X;`) et d'inclure le module "`X.h`" seulement dans le corps de la classe que vous êtes en train d'écrire. Cela gagne du temps à la compilation, notamment si votre système de fichier est un peu lent.

N'oubliez pas de vérifier votre conception objet lorsqu'un cas d'inclusion réciproque apparaît. Souvent, une modification légère de la conception permet d'éviter le problème : création de fonctions externes plutôt que définition d'une méthode, utilisation d'une 3ème classe intermédiaire de conversion, etc.

1.12 Pointeur `this`

Dans le corps d'une méthode d'une classe, vous disposez d'un moyen pour accéder à l'objet courant (ie soi-même) : le pointeur `this`. Si vous êtes dans le corps d'une méthode d'une classe dénommée `A`, alors `this` est un pointeur (constant) de `A`. Ce pointeur stocke l'adresse de l'objet courant.

Si on reprend l'exemple de la classe `Point`, alors on dispose de deux moyens pour accéder aux attributs d'un `Point` dans une de ses méthodes :

```
void Point::affiche() const
{
  cout << " x=" << my_x        // acces direct
       << " y=" << this->my_y  // acces via le pointeur 'this'
       << endl;
}
```

En réalité, la première écriture n'est qu'un raccourci offert par le C++ de la deuxième écriture. Le compilateur fera exactement le même code.

Le pointeur `this` est notamment utilisé dans l'opérateur d'affectation pour savoir si l'objet reçu en paramètre n'est pas en fait soi-même. C'est le cas dans une ligne du genre "`a = a;`".

2 Associations : agrégation

Les classes peuvent être reliées structurellement. Certaines relations sont particulièrement utiles en programmation orientée objet, notamment l'agrégation et l'héritage.

2.1 Agrégation

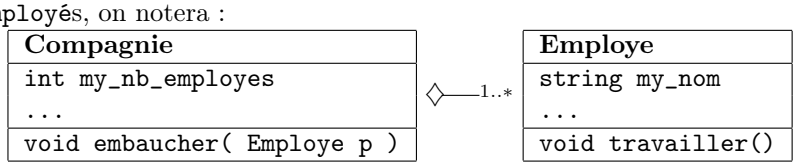
L'*agrégation* permet de définir qu'un objet est l'assemblage d'un ou plusieurs sous-objets. Cette relation n'est pas propre à la programmation orientée objet puisque, même en programmation impérative, on peut écrire qu'une entité est composé d'une ou plusieurs sous-entités. La différence essentielle réside dans le fait que c'est l'objet qui *contient* les sous-objets qui va communiquer lui-même avec ses sous-parties. Pour définir une relation d'agrégation entre des objets, il faut respecter les règles suivantes :

1. les parties doivent avoir une relation structurelle ou fonctionnelle au tout dont elles sont les constituants. L'agrégation peut être utilisée dans les cas suivants (par exemple) :
   (a) pour les objets qui sont les parties d'un assemblage. Le tout a un comportement global cohérent. Chaque constituant conserve un fonctionnement propre. Exemple : un clavier fait partie d'un ordinateur.
   (b) pour les objets qui forment les matériaux définissant ensembles un objet. Chaque constituant n'existe plus vraiment en tant que tel mais fait partie d'un tout. Exemple : le pain est fait de farine, d'eau et de levure.
   (c) pour les objets qui sont des portions d'un objet plus conséquent. Exemple : une coupe dans une image médicale 3D.
   (d) pour les objets qui sont en relation spatiale ou géographique d'inclusion avec un autre objet. Exemple : Yosemite fait partie de la Californie.
   (e) pour les objets qui collectés et ordonnés définissent un tout cohérent. Exemple : un voyage est composé de plusieurs trajets.
   (f) pour les objets contenus dans un conteneur quelconque, sans être forcément du même champ sémantique. Exemple : les employés comme les ressources matérielles font partie d'une entreprise.
2. Il faut que la relation d'agrégation respecte les contraintes suivantes :

   antisymétrie : si un objet A fait partie d'un objet B, alors un objet B ne fait pas partie d'un objet A, même indirectement. Si tel n'est pas le cas, cela veut dire qu'on est en présence d'une simple association.

transitivité : si un objet A fait partie d'un objet B, et que B fait partie d'un objet C, alors A doit faire partie d'un objet C dans tous les cas. Sinon, c'est que les champs sémantiques des objets sont mal définis. (Bref, ce que fait exactement chaque objet est toujours flou dans votre esprit.)

En UML, on note l'agrégation dans les diagrammes de classes avec une relation non symétrique terminée du côté de l'objet agrégeant par le symbole ♢. Si une `Compagnie` possède un certain nombre d'`Employés`, on notera :

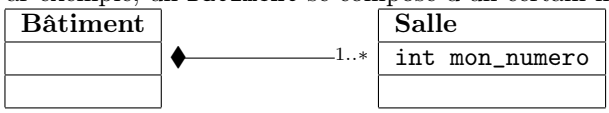


2.2 Agrégation et composition

La *composition* est un cas particulier de l'agrégation, qui implique une inclusion plus forte de l'objet agrégé dans l'objet agrégeant. Alors qu'un même objet peut être agrégé par plusieurs objets différents, un même objet ne peut pas être "composé" (ou partagé) par plusieurs objets. Il y a de plus une notion de durée de vie commune entre un composant et ses composés : les composés d'un composant sont en général détruits en même temps que le composant.

En UML, on utilise la même notation pour la composition que pour l'agrégation, sauf que la pointe ♢ est noircie en ♦.

Par exemple, un `Bâtiment` se compose d'un certain nombre de `Salles` :



2.2.1 Implémentation en C++ de l'agrégation

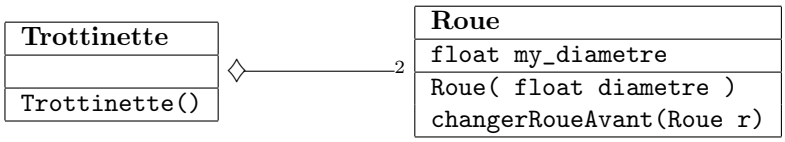
Une relation d'agrégation peut être définie de deux manières différentes en C++ :

1. sous forme d'un attribut de la classe *agrégeante* dont le type est celui de la classe *agrégée*. L'objet agrégé fait donc partie de l'espace mémoire reservée pour l'objet. Cette forme est utilisée lorsque l'objet agrégé n'est pas partagé par plusieurs objets différents. Elle apparaît donc souvent dans le cas d'une composition. Cette écriture est très pratique pour la relation de composition en C++ car le langage C++ impose la durée de vie commune au composant et à ses composés : à la construction d'un objet composant, ses composés sont construits juste avant ; à la destruction d'un objet composant, ses composés sont détruits juste après.

   A noter que lorsque l'on ne connaît pas à l'avance le nombre d'objets agrégés/composés, on ne peut définir un attribut de taille variable en C++ : on utilisera donc soit un pointeur pour faire de l'allocation dynamique, soit un attribut conteneur (genre une liste, un vecteur, etc., cf STL) pour implémenter cette agrégation.

2. sous forme d'un attribut de la classe *agrégeante* dont le type est un pointeur vers la classe *agrégée*. L'objet agrégé ne fait donc pas partie de l'espace mémoire reservé pour l'objet. Son espace doit être alloué et désalloué dynamiquement. Cette forme est utilisée lorsque l'objet agrégé risque d'être partagé par plusieurs objets différents en même temps ou au cours de sa durée de vie. Cette écriture sert aussi pour des compositions où le composé peut être de différent types.

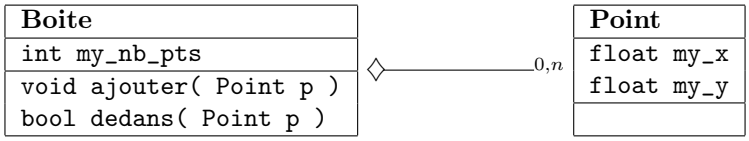
L'exemple suivant montre qu'une trottinette a deux roues :



```
// fichier Roue.h

class Roue {
...
  Roue( float diametre );
...
private:
  float my_diametre;
...
};
```

```
// fichier Trottinette.h
class Trottinette {
...
  Trottinette();
  void changerRoueAvant( Roue r );
...
private:
  Roue my_roue_avant;
  Roue my_roue_arriere;
};
```

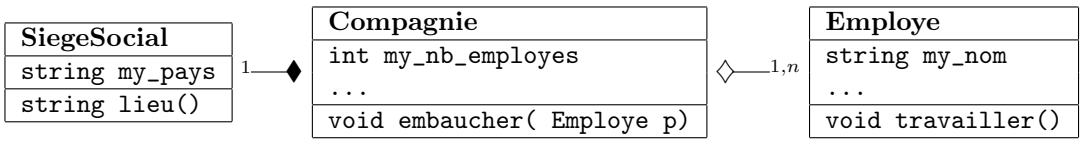
Une boîte englobante contient un certain nombre de points :




```
// fichier Point.h
class Point {
...
private:
  float my_x;
  float my_y;
...
};
```

```
// fichier Boite.h
class Boite {
...
  void ajouter( Point p );
  bool dedans( Point p );
...
private:
  int my_nb_pts;
  Point* my_pts;
};
```

Une compagnie possède un certain nombre d'employés et un siège social :




```
// fichier SiegeSocial.h
class SiegeSocial {
...
  string lieu();
...
private:
  string my_pays;
...
};
```

```
// fichier Compagnie.h
class Compagnie {
...
  void embaucher( Employe p );
...
private:
  int my_nb_employes;
  Employe* my_employes;
  SiegeSocial my_siege;
};
```

```
// fichier Employe.h
class Employe {
...
  void travailler();
...
private:
  string my_nom;
};
```

2.3 Construction/destruction d'un objet agrégeant des sous-objets

On se place dans le cas où l'agrégation a été réalisée en définissant les sous-objets comme attributs de l'objet englobant.

2.3.1 Construction

Si une classe `X` se compose de sous-objets, alors l'instanciation d'un objet de type `X` provoque d'abord l'instanciation des sous-objets de `X` (et donc des sous-objets des sous-objets de `X`, etc). Il est possible de définir quels sont les constructeurs appelés pour chacun des sous-objets avec la notation " `:` " dans le *corps* des constructeurs. Par exemple, pour une trottinette, on pourrait écrire son constructeur comme :

```
Trottinette::Trottinette()
  : my_roue_avant( 10 ),  // 10 cm
    my_roue_arriere( 10 ) // 10 cm aussi
{ // Ici la trottinette et ses composes sont instancies.
  ...
}
```

NB : attention, la notation " `:` " est aussi utilisée pour la relation d'héritage.

2.3.2 Destruction

Lors de la destruction d'un objet de classe `X` contenant des sous-objets, le destructeur de cet objet est d'abord appelé, puis les destructeurs de ses sous-objets sont appelés (en général dans l'ordre de définition).

Exercice 5 : Si on suppose que les constructeurs et destructeur des classes affiche une trace à l'écran du genre :

```
XXX::XXX()            // constructeur par defaut de XXX
YYY::YYY(const YYY &) // constructeur par copie de YYY
ZZZ::~ZZZ()           // destructeur de ZZZ
```

Qu'afficheront les lignes suivantes ?

```
{
  Trottinette t;
  Roue jolie_roue( 12 );
  t.changerRoueAvant( jolie_roue );
}
```

3 Généralisation et héritage

3.1 Un peu de définitions et de vocabulaire

Voilà quelques définitions relatives à la généralisation et l'héritage telles qu'on peut les trouver dans des livres d'analyse et conception orientée objet (c'est-à-dire indépendamment de tout langage de programmation objet).

Généralisation : "mécanisme qui, à une classe origine, dite sous-classe, fait correspondre une classe plus générale, dite super-classe." (Les Objets, M. Bouzeghoub - G. Gardarin - P. Valduriez).

Héritage : "mécanisme de transmission des propriétés (champs et méthodes, attributs et opérations, données et fonctions membres) d'une classe vers une sous-classe (ou classe dérivée)." (Les Objets, M. Bouzeghoub - G. Gardarin - P. Valduriez).

"La généralisation et l'héritage sont des abstractions puissantes qui permettent de partager les points communs entre les classes tout en préservant leurs différences" (OMT, J. Rumbaugh).

3.2 Héritage

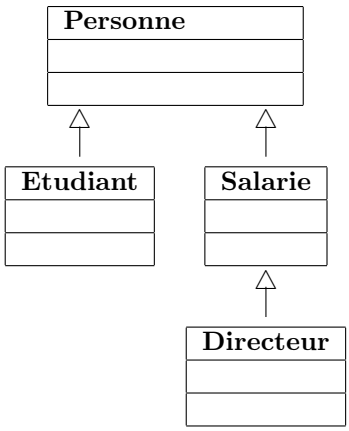
L'*héritage* est l'outil qui va nous permettre d'exprimer entre nos classes une relation de type "est une sorte de". Cet outil est très puissant, car grâce à lui, nous pourrons déclarer des classes très générales, puis progressivement "spécialiser" ces classes. Cette *spécialisation* est aussi une "extension" ou une "dérivation" de la classe de base. Les opérations de haut niveau et communes à un grand nombre d'objets différents sont réalisées dans les objets situés haut dans la hiérarchie. Les opérations plus spécifiques sont réalisées à un niveau plus bas de la hiérarchie, i.e. dans les classes spécialisées ou dérivées, là où il y a suffisamment d'information pour les faire. Les opérations sont donc réparties dans la hiérarchie d'héritage entre les objets. Le découpage en opération se fait du général au spécifique. Le fait de partir d'une classe de base générale permet donc deux démarches fondamentales intervenant à deux niveaux différents dans le processus de développement :

- `niveau conception :` exprimer directement dans le code des concepts ayant un haut niveau d'abstraction, ce qui favorise l'approche descendante.
- `niveau codage :` coder quelques fonctions ou données qui seront "réutilisées" dans les classes dérivant de la classe de base.

Remarque : il est important de considérer en priorité les arguments de conception avant de définir une relation d'héritage. La réutilisation du code lorsqu'on en sera au codage doit être la conséquence d'une bonne conception, pas l'inverse.

3.3 Hiérarchie d'héritage

Plutôt que de réimplémenter des fonctionnalités existantes déjà dans une classe A, une classe B peut intégrer des données et fonctions membres de la classe A. On dit que B *hérite* de A, que B *spécialise* A, que A *généralise* B. B est une *sous-classe* de A.
Bien sûr, plusieurs classes peuvent intervenir et former ainsi ce qu'on appelle une *hiérarchie d'héritage*. Lorsque chaque classe hérite d'une seule classe, on parle d'*héritage simple*, le graphe d'héritage est un arbre.
Lorsqu'une classe hérite de plusieurs classes, on parle d'*héritage multiple*, le graphe d'héritage (si vu comme non-orienté) peut contenir des boucles et plusieurs problèmes peuvent survenir (homonymies, ambiguïté des attributs). On ne s'en préoccupera pas ici. De plus, il est peu de cas où l'héritage multiple se justifie complètement. Certains langages comme JAVA limitent d'ailleurs l'héritage multiple.



3.4 Différences agrégation/héritage

Attention de ne pas confondre relation d'héritage entre classes avec l'association et l'agrégation :

- l'héritage indique une relation "est un" ; par exemple un enseignant est un utilisateur ;
- l'association (souvent binaire) entre classes est caractérisée par un verbe décrivant les liens entre les classes ; une caractéristique importante d'une association est sa multiplicité (voir UML). Par exemple un enseignant peut enseigner plusieurs matières, un étudiant est inscrit dans un établissement, ...)
- l'agrégation est un cas particulier d'association représentant la relation "partie de". Par exemple, un étudiant fait partie d'un groupe.

3.5 Implémentation en C++

Nous ne traiterons ici que l'héritage simple. Voici un exemple en C++ pour introduire la syntaxe et les notions.

```
class A {
  public:
     A();
     ~A();
     void f();
     void g();
     int n;
  private:
     int p;
};
```

```
// La classe B herite de la classe A
class B : public A {  // heritage public
  public:
     B();
     ~B();
     void h();
  private:
     int q;
};
```

```
int main() {
  B b;
  b.h();    // OK, fct membre de B
  b.f();    // OK, fct membre (public) heritee de A
  b.n = 3;  // OK, donnee membre (public) heritee de A
  b.p = 5;  // ERREUR, donnee membre (private) heritee de A
            // interdit aussi dans une fonction membre de B...
}
```

Exercice 6 : Indiquez les lignes qui poseront problème dans l'extrait de code ci-dessous :

```
// Fichier B.cxx
void B::h() {
  f();
  A::f();
}
```

```
// Fichier A.cxx
void A::g() {
  h();
  B::h();
  this->f();
}
```

3.6 Visibilité des membres d'une classe

Un membre privé (section `private`) n'est accessible que par les méthodes de la classe qui la définit. Une classe peut spécifier les membres disponibles à ses classes dérivées (section `protected`). La section `public` est accessible à tout le monde.

La classe B peut hériter de la classe A de trois façons différentes :

`class B :` <*contrôle d'accès*> `A`

où <*contrôle d'accès*> =

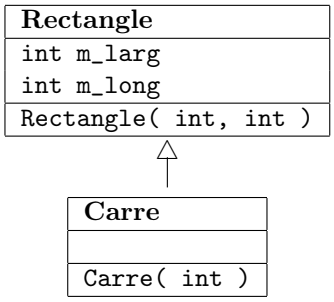
- `public` contrôle d'accès sans changement : données publiques restent publiques, protégées restent protégées, privées restent privées ;
  (L'héritage public est utile lorsque vous voulez rajouter un nouveau comportement à un objet existant, sans vous préocupper du fait que l'objet puisse être utilisé aussi sous sa forme initiale. Cette forme d'héritage représente la majeure partie des cas.)
- `protected` données publiques deviennent protégées, protégées restent protégées, privées restent privées ;
  (L'héritage protégé est employé lorsque vous voulez réutiliser le comportement d'un objet pour le redéfinir/compléter, mais que vous craigniez que votre redéfinition/complétion mette en danger l'usage de l'objet vu sous sa forme plus abstraite. Schématiquement, vous autorisez l'usage de l'objet sous sa forme abstraite à vous-même et à vos éventuelles classes dérivées. Cela évite qu'un utilisateur quelconque perturbe le comportement de votre objet dérivé en se servant de ses propriétés héritées.)
- `private` données publiques et protégées deviennent privées, privées restent privées. Ainsi toute classe qui dérive de cette classe dérivée ne peut se servir de la forme première de la classe. De même, un utilisateur de `B` ne peut pas se servir de ses propriétés qui viennent de `A`. On parle souvent d'héritage pour implémentation.
  (L'héritage privé permet d'utiliser les propriétés d'une super-classe dans une sous-classe sans qu'un utilisateur puisse se servir des propriétés de la super-classe. Par exemple, on peut écrire une pile à l'aide d'une liste. Un moyen commode est donc de dériver de liste. Mais on ne veut pas qu'un utilisateur se serve de notre pile comme une liste : on utilise alors l'héritage privé.)

Une classe dérivée peut donc accéder aux membres hérités publics ou protégés mais ne peut accéder aux membres privés hérités.

Résumé : les champs privés hérités ne sont manipulables qu'à l'aide des méthodes publiques ou protégées héritées. Les classes dérivées n'ont donc pas plus le droit de violer l'encapsulation de la classe de base que toute autre classe. Le mécanisme `protected` permet de définir des données accessibles par toute classe dérivée mais considérées comme privée en dehors de la relation d'héritage.

3.7 Constructeurs dans une hiérarchie d'héritage

Lorsqu'une classe B hérite d'une classe A, l'instanciation d'un objet de type B provoque l'instanciation d'un objet de type A (un B est un A). Dans le corps des constructeurs de la classe B, vous devez préciser quel est le constructeur de la classe A qui doit être appelé à l'instanciation de B. Similairement à l'appel des constructeurs des attributs, on utilise la notation ':' pour appeler le constructeur de la super-classe. L'exemple ci-dessous illustre ce mécanisme.



fichier `Carre.h`

```
class Carre : public Rectangle {
...
  Carre( int cote );
...
};
```

fichier `Carre.cxx`

```
Carre::Carre( int cote )
  : Rectangle( cote, cote )
  // appel constructeur de Rectangle
{
 ...
}
```

3.8 Destructeur dans une hiérarchie d'héritage

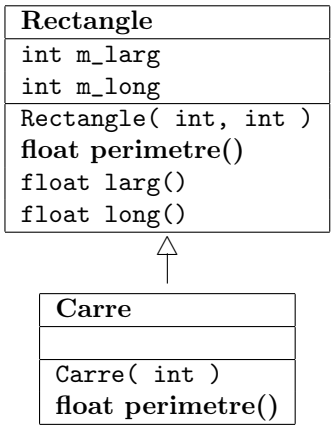
Si B hérite de A, lorsqu'une instance de B est détruite, l'instance de A correspondante est aussi détruite automatiquement. L'ordre de destruction est la classe dérivée puis la super-classe. Les appels des destructeurs respectifs sont fait automatiquement. Par exemple, si C hérite de B, une instance de C détruite provoquera l'appel du destructeur de C, puis du destructeur de B, puis du destructeur de A.

3.9 Méthodes héritées et substitution

*Attention, le terme surcharge (*overloading*) est employé en UML pour désigner la substitution (*overriding*).*

Il est parfois intéressant de redéfinir le comportement d'une méthode d'une classe ancêtre dans la classe dérivée. Cela permet de réaliser certaines opérations plus rapidement par exemple. Lorsqu'on redéfinit une méthode dans une classe dérivée (en définissant une méthode de même signature qu'une méthode héritée), on dit qu'on réalise la *substitution* de cette méthode.

Sur l'exemple suivant, on substitue la méthode `perimetre` de la classe `Rectangle` dans la classe `Carre` pour réaliser l'opération plus efficacement :



fichier `Carre.h`

```
class Carre : public Rectangle {
...
  Carre( int cote );
  float perimetre();
...
};
```

fichier `Carre.cxx`

```
float Carre::perimetre()
{
  return 4*long();
}
```

Un fichier `.cxx`

```
Rectangle r( 100, 50 );
Carre c( 30 );
cout << r.perimetre() // appel perimetre de 'Rectangle'.
     << c.perimetre() // appel perimetre de 'Carre'.
     << endl;
```

3.10 Portée des méthodes

On voudrait parfois appeler une méthode définie à un niveau précis dans une hiérarchie d'héritage. Par exemple, on voudrait appeler le `perimetre` de `Rectangle` même si on manipule un `Carre`. On utilisera alors l'opérateur de résolution de portée '`::`'. On pourra ainsi écrire :

```
cout << c.perimetre()            // appel perimetre de 'Carre'.
     << c.Rectangle::perimetre() // appel perimetre de 'Rectangle'.
     << endl;
```

C'est notamment utile lorsque vous substituez une méthode dans une classe dérivée et que, dans le corps de cette méthode substituée, vous voulez rappeler le comportement de la méthode définie dans une super-classe.

3.11 Polymorphisme, classes abstraites

"Le polymorphisme - la possibilité pour un programmeur de traiter plusieurs formes d'une classe comme si elles n'étaient qu'une - est un puissant mécanisme d'abstraction qui protège les programmeurs des détails de réalisation des classes dérivées" (Programmation avancée en C++ - James O. Coplien).

En C++, on peut manipuler tout objet de type `A` soit comme un objet de type `A` (normal !), soit comme un objet de type `B` où la classe `B` est une superclasse quelconque de `A`. Si on dispose d'une collection d'objets dont les types d'instanciation sont des sous-classes d'une classe `A`, on peut manipuler tous ces objets de façon uniforme en les considérant comme des objets de type `A`. Mieux, on peut quand même spécialiser certains comportements suivant le type d'instanciation de chaque objet. En d'autres termes, l'utilisation d'objets distincts d'une même hiérarchie est homogène même si le comportement de ces objets reste spécifique. On parle de *polymorphisme*.

3.11.1 Polymorphisme : désignation d'un objet à l'aide d'un pointeur ou d'une référence de type plus abstrait

Tout objet d'une classe dérivée peut être traité et utilisé comme un objet de sa classe de base (en fait comme n'importe lequel de ses ancêtres). Si `B` hérite de `A`, on peut ainsi affecter un objet de type `B` à un objet de type `A`, passer un objet de type `B` par valeur ou réference dans une fonction ou

méthode qui attend un argument de type `A`. On peut aussi utiliser un pointeur vers `A` pour pointer vers un objet instancié en tant que `B`.

Par exemple :

```
A a;
B b; // B est une classe derivee de A.
a = b; // correct, seule la partie A de b
       // fait l'objet de l'affectation.
b = a; // incorrect, un A n'est pas un B.
A* pa = &a; // correct bien sur.
pa = &b;    // correct aussi.
B* pb = &a; // erreur.
```

```
void fct( A & obj ) {...}
...
fct( a ); // correct.
fct( b ); // correct aussi.
```

3.11.2 Polymorphisme : méthodes virtuelles

Une opération substituée (surchargée) peut donc prendre plusieurs formes ou implémentation en fonction du type d'objet auquel elle s'applique. L'opération est dite alors *polymorphe*. En C++, on parle alors de méthode *virtuelle* (méthode à laquelle on rajoute le mot-clé `virtual` dans la définition).

Considérons l'exemple suivant qui se base sur les `Carre`s et `Rectangle`s :

```
1 int main()
2 {
3   Rectangle r( 100, 50 );
4   Carre c( 30 );
5   Rectangle*  ptr_r;
6
7   if ( condition )  ptr_r = &r;
8   else              ptr_r = &c;
9   cout << ptr_r->perimetre() << endl;
10}
```

A la ligne 5, nous définissons un pointeur `ptr_r`. Comme vu dans un paragraphe précédent, un pointeur sur une classe de base peut au cours de l'exécution pointer sur n'importe quel objet dérivé. Les affectations des lignes 7 et 8 sont donc valides.

Nous pouvons alors nous interroger pour savoir quelle est la fonction `perimetre` exécutée à la ligne 9.

Dans le cas d'une liaison statique (décidée par le compilateur) la seule fonction qui couvre tous les cas est la fonction `perimetre` de la classe de base (i.e. `Rectangle::perimetre`). Ce cas a peu d'intérêt car nous avons défini une fonction `Carre::perimetre` que l'on aurait souhaité exécuter dans le cas où `ptr_r` pointe sur une instance de type `Carre`.

En demandant que la fonction `perimetre` soit définie *virtuelle*, on impose une *liaison dynamique*. La fonction `perimetre` à appeler sera alors déterminée à l'exécution en fonction du type de l'objet. Sur l'exemple précédent : si la condition est vraie alors `ptr_r` pointe sur `r`, la fonction `Rectangle::perimetre` est exécutée, si la condition est fausse, `ptr_r` pointe sur `c`, la fonction `Carre::perimetre` est exécutée.

Liaison dynamique : mécanisme de sélection de code d'une opération à l'exécution en fonction de la classe d'appartenance de l'objet.

En C++, voilà le code des classes `Rectangle` et `Carre` pour imposer la liaison dynamique :

fichier `Rectangle.h`

```
class Rectangle {
...
  virtual float perimetre();
...
};
```

fichier `Carre.h`

```
class Carre : public Rectangle {
...
  Carre( int cote );
  virtual float perimetre();
  // virtual non obligatoire, mais
  // recommande.
...
};
```

Un fichier `.cxx`

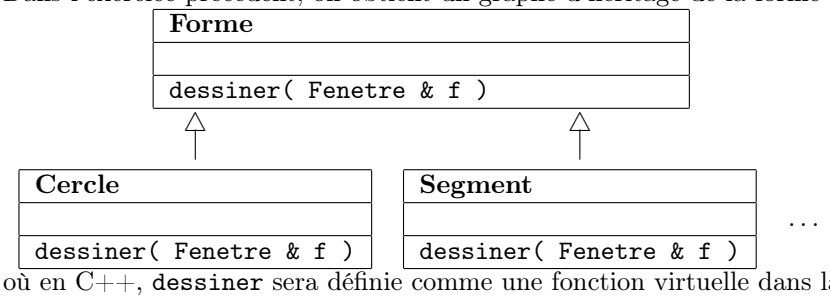
```
Rectangle r( 100, 50 );
Carre c( 30 );
Rectangle*  ptr_r;

if ( condition )  ptr_r = &r;
else              ptr_r = &c;
cout << ptr_r->perimetre() << endl;
// appel le 'perimetre' de 'Carre'
// ou 'Rectangle' selon les cas.
```

On remarque que ni les corps des classes ni les fonctions extérieurs à la classe ne sont pas modifiés.

Exercice 7 : Un programme manipule différentes formes géométriques (segment de droite, carré, cercle, etc). Ces différentes formes sont stockées dans une liste. Donnez un graphe d'héritage. Quelles fonctions peuvent être substituées, définies virtuelles ? Fournir un extrait de code permettant d'afficher l'ensemble des formes, calculer le périmètre et l'aire de toutes les formes, etc.

3.11.3 Classes abstraites, méthodes virtuelles pures

Dans l'exercice précédent, on obtient un graphe d'héritage de la forme :



où en C++, `dessiner` sera définie comme une fonction virtuelle dans la classe `Forme`.

Fournir le code des fonctions `dessiner` de la classe `Segment` ou `Cercle` ne pose pas de problème particulier.

Mais quelle code pour la fonction `dessiner` de la classe `Forme` ? En fait, il n'y aura jamais création d'une instance de la classe `Forme`. Seuls les objets dérivées sont instanciés. La classe `Forme` est une classe qui permet de définir le comportement commun des sous-classes `Segment`, `Cercle`, etc. C'est ce qu'on appelle une *classe abstraite*.

Une telle classe comprend des méthodes virtuelles sans code qui sont dites *méthodes virtuelles pures* (ou *méthodes abstraites*). Evidement dès qu'une classe possède une méthode virtuelle pure, il devient alors interdit d'instancier un objet de cette classe. Une telle classe peut bien sûr posséder des fonctions complètement définies (avec code), fonctions qui seront héritées dans les sous-classes et qu'il sera donc inutile de réécrire dans celles-ci.

On écrira les méthodes virtuelles pures en rajoutant le symbole `= 0` à la définition de la méthode. Pour la classe `Forme`, cela donne :

fichier `Forme.h`

```
class Forme
{
  ...
  virtual void dessiner(Fenetre & f) = 0;
  // Notez le '= 0' qui transforme 'Forme' en classe abstraite.
  ...
};
```

La notation `= 0` permet en C++ d'indiquer que la fonction `dessiner` est une méthode pure et donc que la classe `Forme` est une classe abstraite.

Remarque : si vous écrivez une classe héritant d'une classe abstraite, mais qui ne définit pas de corps à une (ou plus) méthode virtuelle pure héritée, alors cette nouvelle classe est toujours une classe abstraite, non instanciable.

L'exemple suivant montre que le compilateur refuse de compiler un programme où une classe abstraite est instanciée :

fichier `essai.cxx`

```
class A {
public:
  virtual void fct() = 0;
};

int main( int argc, char** argv )
{
  A a;
}
```

Un shell

```
=> g++ essai.cxx
essai.cxx: In function `int main (int, char **)':
essai.cxx:8: cannot declare variable `a' to be of type `A'
essai.cxx:8:   since the following virtual functions are abstract:
essai.cxx:3:    void A::fct ()
=>
```

Les classes abstraites représentent un puissant outil d'analyse et de conception, car elles permettent de mettre en évidence des processus et des utilisations communs de différentes classes, sans préjuger d'une quelconque implémentation. Au niveau du codage, elles permettent d'unifier le traitement d'objets différents mais qui répondent à des stimuli communs. L'exemple des `Forme`s géométriques est ultra classique, mais elle est effectivement très élégante, à la fois d'un point de vue conception et codage. Les classes abstraites sont aussi très utilisées pour spécifier le(s) comportement(s) communs des objets d'une interface (eg., se réafficher, réagir à un click souris, se redimensionner), puis des objets dérivés spécialisent certains comportements (eg., différents styles graphiques).

Table des matières