4. Fonction virtuelle, polymorphisme, classe abstraite

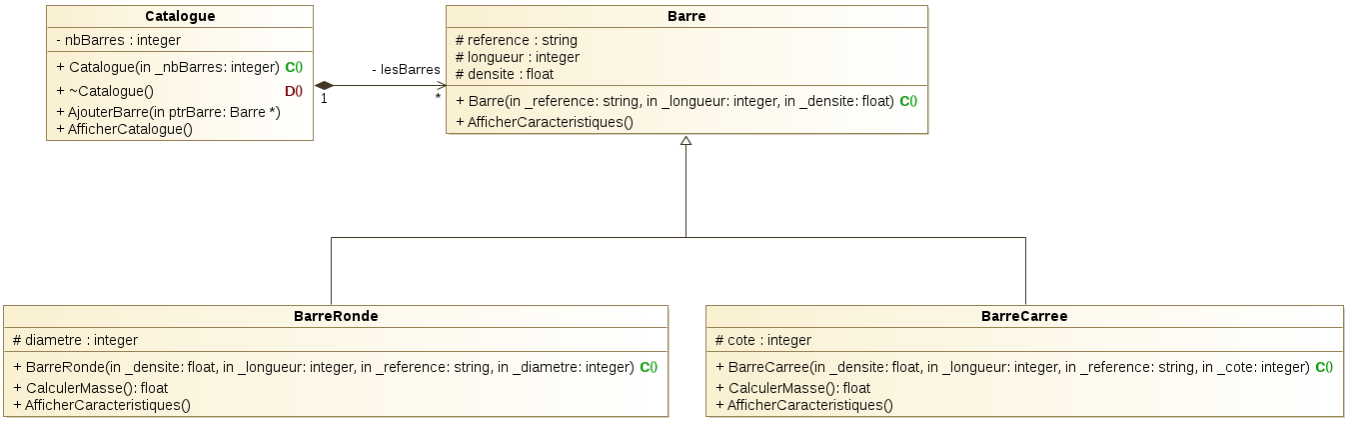
Note au lecteur

Pour aborder ce chapitre, la notion d'héritage doit parfaitement être assimilée. En effet, les concepts de fonctions virtuelles et de polymorphisme s'appuient sur cette relation d'héritage.

4.1. Introduction

Le diagramme de classes suivant représente une hiérarchie de classes, les Classes **BarreRonde** et **BarreCarree** sont des sortes de la classe **Barre**.

La classe **Catalogue** contient physiquement des instances de la classe **Barre** ou ces héritiers.



La classe Catalogue doit permettre le stockage et la consultation des différentes barres fabriquées dans l'usine. Une version partielle tel qu'elle est représentée dans le diagramme de classes est donnée ici :

Déclaration de la classe Catalogue *catalogue.h*

```
#ifndef CATALOGUE_H
#define CATALOGUE_H

#include "barre.h"

class Catalogue
{
private:
    Barre **lesBarres;  // pour la création d'un tableau de pointeurs de barre
    int index;          // index de la prochaine case libre du tableau
    const int nbBarres; // nombre maxi de barres dans le tableau
public:
    Catalogue(const int _nbBarres);
    ~Catalogue();
    bool AjouterBarre(Barre *ptrBarre);
    void AfficherCatalogue();
};

#endif // CATALOGUE_H
```

Dans cet exemple, chaque nouvelle barre fait l'objet d'une allocation dynamique. Pour les rassembler sous la forme d'un catalogue ou une collection, il est nécessaire d'utiliser un tableau de pointeurs. Le tableau de pointeur est également alloué dynamiquement en fonction du nombre de barres à stocker, d'où cette déclaration avec deux étoiles : **Barre **lesBarres;**

Pour ce qui concerne l'implémentation de la classe **Catalogue**, voici le détail :

Implémentation de la classe Catalogue *catalogue.cpp*

```
#include <iostream>
#include "catalogue.h"

using namespace std;

Catalogue::Catalogue(const int _nbBarres):
    nbBarres(_nbBarres)
{
    lesBarres = new Barre *[nbBarres];
    index = 0;
}

Catalogue::~Catalogue()
{
    delete[] lesBarres;
}

bool Catalogue::AjouterBarre(Barre *ptrBarre)
{
    bool retour = true;
    if (index < nbBarres)
          lesBarres[index++] = ptrBarre ;
    else
        retour = false;
    return retour;
}

void Catalogue::AfficherCatalogue()
{
    for (int indice = 0; indice < index ; indice++)
    {
        lesBarres[indice]->AfficherCaracteristiques();
        cout << lesBarres[indice]->CalculerMasse() << endl;
        //erreur de compilation pour cette dernière ligne
    }
}
```

La variable **lesBarres[indice]** utilisée dans les fonctions **Catalogue::AfficherCatalogue()** contient des pointeurs sur des objets de type classe **Barre**. Hors l'objet réel contenu dans les différentes cases **lesBarres[indice]** est de type pointeur sur **BarreRonde** ou **BarreCarree**, le compilateur est donc actuellement incapable de connaître le type réel de l'objet qui n'est défini qu'à l'exécution. On dit que l'appel de fonction est donc résolu **statiquement** à la compilation, le résultat ne sera pas celui attendu.

Pour la dernière ligne, le compilateur ne veut pas compiler, il n'y a pas de fonction **CalculerMasse()** dans la classe **Barre**.

Pour un fonctionnement correct, il faudrait que ce soit le type réel de l'objet manipulé qui détermine la fonction à appeler, et non le type générique du pointeur contenu dans **lesBarres[indice]**. Ce mécanisme ne peut donc être effectué qu'au cours de l'exécution.

À retenir

Dans une hiérarchie de classes, il est possible de convertir un pointeur de classe dérivée en un pointeur de classe de base pour réaliser une collection. L'inverse est également possible, mais cela peut s'avérer dangereux, car la classe dérivée peut avoir des membres qui ne sont pas dans la classe de base, dans ce cas il est nécessaire de procéder à un transtypage pour assurer la conversion. Ici, seuls les membres de la classe de base seront accessibles.

En C++, sans précision particulière, l'appel de fonction est résolu au moment de la compilation et de l'édition de liens de manière statique.

4.2. Fonctions virtuelles

Pour résoudre ce problème inhérent au typage statique des objets, les concepteurs du C++ ont introduit la notion de fonction virtuelle. Lorsqu'une fonction est déclarée virtuelle en plaçant le mot-clé **virtual** devant sa déclaration. Les appels à une telle fonction ou à n'importe laquelle de ses redéfinitions dans des classes dérivées sont "résolus" au moment de l'exécution. On parle de résolution dynamique de liens ou de typage ***dynamique*** des objets ou encore de ***polymorphisme d'héritage***.

Dans l'exemple précédent, il suffit de déclarer la classe **Barre** de la manière suivante :

Déclaration de la classe Barre *barre.h*

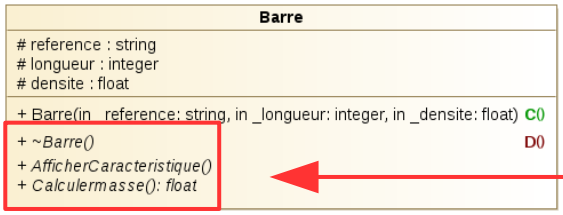
```
#ifndef BARRE_H
#define BARRE_H

#include <string>

using namespace std;

class Barre
{
protected:
    string reference;
    int longueur;
    float densite;
public:
    Barre(const string _reference, const int _longueur,const float _densite);
    virtual ~Barre();
    virtual void AfficherCaracteristiques();
    virtual float CalculerMasse() {return 0;}
};

#endif // BARRE_H
```

Par rapport à ce qui aurait pu être déduit du premier diagramme de classe, on constate les modifications à apporter.

- Création d'un destructeur virtuel, cela permet d'être certain d'appeler le bon destructeur, ci-nécessaire.
- Rendre la méthode **AfficherCarcacteristiques()** virtuelle, pour appeler la bonne méthode en fonction type d'objet pointé.
- Créer une méthode **CalculerMasse()** virtuelle qui n'existait pas précédemment pour ne pas avoir d'erreurs de compilation. Ici, la fonction a été déclarée et définie dans le fichier d'en-tête, on parle de déclaration **inline**. Elle ne fait que retourner 0, en effet il n'est pas possible de calculer la masse d'une barre pour l'instant, il nous manque des informations pour déterminer la section de la barre.

Aucune autre modification n'est nécessaire.

Nouvelle représentation UML de la classe :



Les méthodes virtuelles sont représentées en italique dans la classe.

À retenir

Le mot clé **virtual** peut être employé qu'une fois pour une fonction donnée dans la classe de base. Il est cependant préférable, pour la clarté, de répéter **virtual** dans les déclarations des classes dérivées.

Une classe dérivée n'est pas obligée de redéfinir une fonction virtuelle de sa classe de base.

Toutes les classes dérivées qui redéfinissent une fonction virtuelle doivent utiliser le même prototype que celui défini dans la classe de base.

4.3. Avantage des fonctions virtuelles

Le mécanisme de fonction virtuelle, associé à celui de l'héritage, fournit un code encore plus ***réutilisable et extensible***.

Grâce à l'héritage seul, le code écrit pour la classe de base est réutilisable par les classes dérivées. Dans l'exemple, les classes **BarreRonde** et **BarreCarree** bénéficiaient des propriétés de la classe **Barre** :

- Utilisation du constructeur.
- Utilisation des attributs.
- Utilisation des méthodes éventuelles dans chacune des méthodes des classes dérivées.

Les fonctions virtuelles permettent d'aller plus loin encore, car elles permettent l'écriture de fonctions ***génériques***, fonctions qui s'appliquent de manière uniforme à toute une hiérarchie de classes. Ainsi, pour l'écriture de classe **Catalogue**, il n'est même pas nécessaire de connaître les classes **BarreRonde** ou **BarreCarree**. La connaissance seule de la classe de base **Barre** est suffisante. À chaque classe dérivée de redéfinir les fonctions virtuelles pour qu'elles s'appliquent correctement à son cas particulier.

De plus, pour dériver de nouvelles classes à partir de la classe **Barre**, il n'est même pas nécessaire de recompiler les fichiers sources contenant les classes **Catalogue** et **Barre**. On peut donc définir de nouvelles classes sur lesquelles les méthodes **AfficherCarateristiques()** et **CalculerMasse()** s'appliqueront parfaitement, ***sans modifier, ni même connaître*** le code source des classes de bases.

À retenir

Un constructeur ne peut pas être virtuel, un destructeur peut l'être.

En C++, la règle générale veut que chaque classe qui définit au moins une fonction virtuelle définisse également un destructeur virtuel.

Ceci évite les problèmes, lorsqu'une classe dérivée alloue dynamiquement de la mémoire et libère cette mémoire dans son propre destructeur.

4.4. Fonctions virtuelles pures et Classes abstraites

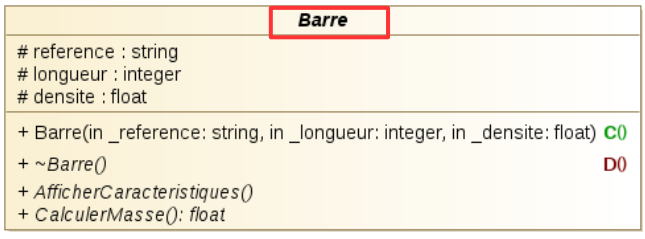
Une fonction virtuelle pure se déclare avec une initialisation à zéro. C'est ce qui aurait pu être fait avec la nouvelle fonction **CalculerMasse( )** de la classe **Barre** :

```
virtual float CalculerMasse() = 0; // fonction virtuelle pure
```

Lorsqu'une classe comporte au moins une fonction virtuelle pure, elle est considérée comme classe **abstraite**, c'est-à-dire qu'il n'est plus possible d'instancier cette classe, mais seulement des objets dérivés.

Dans notre cas, il semble en effet peu raisonnable de déclarer un objet de type **Barre**. La classe **Barre** est une classe générique, elle ne peut représenter correctement une barre en particulier, les caractéristiques de sa forme ne sont pas connues.

Remarque : Le nom de la classe **Barre** devenue abstraite apparaît en caractère italique sous UML

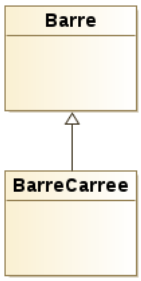


Une classe dérivée qui ne fournit pas une implémentation pour toutes les fonctions virtuelles pures de sa classe de base reste toujours une classe abstraite et ne peut donc pas être directement instanciée.

À retenir

Une classe contenant au moins une méthode virtuelle pure est une classe abstraite. Elle ne peut pas être instanciée et doit faire l'objet d'une dérivation. La classe dérivée doit redéfinir les méthodes virtuelles pures de la classe de base pour être instanciée.

4.5. Application



1. Réalisez un projet C++ en mode console nommé **LesBarres**. Dans un premier temps, ce projet comporte deux classes **Barre** et **BarreCarree**. Déclarez ces deux classes en respectant la hiérarchie proposée. La classe Barre doit être présentée comme dans le paragraphe des fonctions virtuelles.
2. Pour la méthode **BarreCarree::CalculerMasse()**, on rappel la règle de calcul :

   masse = longueur * section * densité et la section d'un carré = coté * coté
3. Réalisez les méthodes **AfficherCaractéristiques()** et complétez les constructeurs et destructeurs pour obtenir l'affichage ci-dessous.
4. On donne le programme principal suivant pour tester

Programme principal — ***lesbarres.cpp***

```
#include <iostream>
#include "barrecarree.h"

using namespace  std;

int main(int argc, char *argv[])
{
    BarreCarree uneBarre("Barre 2x2 en Cuivre", 200, 8.920, 2);
    uneBarre.AfficherCaracteristiques();
    cout << "Le poids de la barre est : " ;
    cout << uneBarre.CalculerMasse() / 1000.0;
    cout << " kg" << endl;
    cout << endl;

    return 0;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Appel du constructeur de Barre
Appel du constructeur de BarreCarree

Référence : Barre 2x2 en Cuivre
Longueur : 200 cm
Densité : 8.92 g/cm3
Coté du carré : 2 cm
Le poids de la barre est : 7.136 kg

Appel  du destructeur de BarreCarree
Appel  du destructeur de Barre
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

5. Procédez de la même manière avec la classe **BarreRonde** et vérifiez son fonctionnement. Pour rappel la section d'un cercle est section = PI * diametre^2 / 4 pour la valeur de Pi, la bibliothèque <math.h> peut être incluse et le nombre Pi est la constante **M_PI**.
6. Ajouter maintenant la classe **Catalogue** qui a été développée précédemment ainsi que la classe Menu développée lors de la première application sur le thème du menu.
7. En vous inspirant de la première application, réalisez le programme principal permettant de gérer le catalogue de barre. Voici des exemples d'écrans attendus :

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
+-----+------------------------+
|  1  | Ajouter Barre carrée   |
|  2  | Ajouter Barre ronde    |
|  3  | Afficher le catalogue  |
|  4  | Quitter                |
+-----+------------------------+

votre choix svp :
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Référence de la barre : Barre carrée de 2x2x200
Longueur en cm : 200
la densité du métal en g/cm3 : 8.5
Coté du carré en cm : 2

Choix n°1
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Référence de la barre : Barre ronde diamètre 2
Longueur en cm : 200
la densité du métal en g/cm3 : 8.5
diamètre de la section en cm : 2

Choix n°2
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Référence : Barre carrée de 2x2x200
Longueur : 200 cm
Densité : 8.5 g/cm3
Coté du carré : 2 cm
Poids de la barre : 6.8 kg

Référence : Barre ronde diamètre 2
Longueur : 200 cm
Densité : 8.5 g/cm3
Diamètre : 2 cm
Poids de la barre : 5.34071 kg

appuyer sur la touche Entrée pour continuer...
Choix n°3
```