Lycée TOUCHARD WASHINGTON

Développement logiciel

Programmation orientée objet : Le langage C++

OBJECTIFS

À l'issue de cette séquence, vous devrez être capable de :

- De créer une classe à partir d'un diagramme UML.
- De réaliser des entrées / sorties en mode console en utilisant les flux.
- D'interagir avec un fichier en utilisant un flux.
- D'instancier un objet, de manière automatique, dynamique et statique.
- D'appeler ses méthodes et de leur passer des paramètres en entrée et/ou en sortie
- De mettre en œuvre un passage de paramètres par valeur, par adresse, par référence.
- De différencier les différentes relations dans un diagramme de classes et de savoir les implémenter en C++.
- De définir des méthodes virtuelles et des classes abstraites.
- De comprendre le concept de template
- D'utiliser la librairie STL en particulier ses conteneurs et ses itérateurs associés.

Niveau de maîtrise attendu pour le BTS Systèmes Numérique option Informatique et Réseaux :

S4. Développement logiciel		IR
S4.6. Programmation orientée objet (Support : C++)	Du C au C++ : références, entrées/sorties (iostream, fstream), polymorphisme, etc.	3
	Définition de classes (encapsulation) et modèle canonique (dit de Coplien)	3
	Instanciation d'objets (new, delete, etc.)	4
	Surcharges d'opérateurs (injection, etc.)	2
	Mécanisme d'héritage	4
	Mécanismes d'agrégation et de composition	4
	Classes abstraites, virtualité	3
	Programmation générique : structure de la STL, conteneurs et itérateurs	2
	Programmation générique : classes paramétrées (template)	2
	Programmation générique : patrons de développement (design patterns)	1

CIEL Version 1.3 (2024)

Sommaire

1. Du C au C++

1.1. Objets et classes

La programmation en C++ diffère de celle en langage C de par son style. En effet, le C++ utilise une collection d'objets comprenant à la fois une structure de données et un comportement, alors que pour le langage C les données et les traitements sont vraiment séparés. L'approche du développement logiciel est fondée sur la modélisation du monde réel.

Les structures de contrôle, boucles, conditions sont identiques de même pour le reste de la syntaxe. En revanche, chaque type devient une classe.

Dissocions maintenant les termes **objet** et **classe** :

Une classe est un type défini par le programmeur ou faisant partie d'une bibliothèque. C'est un ensemble regroupant les mêmes **attributs** ou données membres et disposant des mêmes **méthodes** ou fonctions membres.

Un objet est un élément qui possède sa propre identité, ses propres caractéristiques. Deux objets sont distincts même si tous leurs **attributs** ou **données membres** ont des valeurs identiques. On dit qu'un objet est une **instance** de **sa classe.** C'est l'équivalent d'une variable pour la programmation en langage C.

Attention : une classe ne contient pas d'objets. Une classe aura par défaut, ses données **privées** et ses méthodes **publiques**. C'est le principe de l'**encapsulation**.

La notion de classe sera développée au chapitre 2 : ***Définition de classe***

1.2. Classification des objets

Ce paragraphe s'applique tous les objets, car comme cela a été dit précédemment en C++, tous les types sont des classes.

Les objets **automatiques** sont créés par une déclaration soit dans une fonction, soit au sein d'un bloc. Ils sont créés au moment de l'exécution de la déclaration. Ils sont détruits lorsque l'on sort de la fonction ou du bloc. Les objets automatiques sont détruits dans l'ordre inverse de leur création. Leur durée de vie et leur visibilité sont égales à celle du bloc ou de la fonction où ils sont déclarés. On les associe aux variables locales du langage C.

Les objets **statiques** sont créés par une déclaration située en dehors de toute fonction ou par une déclaration précédée du mot clé ***static***, dans une fonction ou dans un bloc. Ils sont créés avant l'entrée dans la fonction ***main*** et détruits après la fin de son exécution. Ces objets sont accessibles par toutes les méthodes constituant le programme et leur durée de vie est égale à celle de l'application. On les associe aux variables globales du langage C.

Les objets **dynamiques** sont créés par l'opérateur ***new*** et doivent être détruits explicitement par l'opérateur ***delete***. La règle de visibilité est égale à celle du bloc ou de la méthode où ils sont déclarés. Leur durée de vie dépend du programmeur, la mémoire est libérée après l'utilisation de l'opérateur ***delete***.

Les objets **temporaires** sont créés lors de l'appel explicite d'une méthode de classe ou d'un constructeur de classe lors d'un passage par valeur ou d'un retour par valeur d'un objet en argument ou encore lors de l'affectation d'un objet existant sur un nouvel objet créé. Ces objets temporaires permettent, tout simplement, la recopie d'un objet.

1.3. Espace de nommage

Les espaces de nommage ou espaces de noms ont été apportés au C++ pour éviter les conflits entre identificateurs globaux qui pourraient apparaître lors de l'utilisation de nombreuses bibliothèques.

La création d'un espace de nom conduit à réaliser une région déclarative de la forme :

Déclaration d'un espace de nom

```
namespace identifiant
{
}
```

La bibliothèque réalisée est placée entre les accolades.

Par exemple, la librairie standard du C++ est placée dans l'espace de nom ***std***. L'accès à chacun de ses membres doit être précédé du nom de l'espace, ici ***std*****,** et de l'opérateur de résolution de portée **::** comme le montre l'exemple suivant qui déclare une chaîne de caractère de type ***string***. Cet objet remplace, en simplifiant, un tableau de caractères et toutes les fonctions de traitement sur les chaînes du langage C. La taille n'a pas besoin d'être définie à l'avance et elle peut évoluer dynamiquement. De nombreuses méthodes permettent d'agir sur la chaîne. L'ensemble de la classe ***string*** est décrite ici : http://www.cplusplus.com/reference/string/string/.

Utilisation d'un élément de l'espace de nom std

```
#include <string>

int main()
{
    std::string uneChaine ;
    //utilisation de la variable uneChaine

    return 0 ;
}
```

Une autre manière de procéder, lorsqu'il n'y a pas de risque de conflits, est d'indiquer l'utilisation d'un espace de nom de manière implicite.

Utilisation d'un élément de l'espace de nom std de manière implicite

```
#include <string>

using namespace std;

int main()
{
    string uneChaine ;
    //utilisation de la variable uneChaine

    return 0 ;
}
```

En utilisant la directive ***using namespace***, il n'est plus nécessaire de faire précéder les éléments du nom de l'espace de nommage, ici ***std***, et de l'opérateur de résolution de portée **::**

1.4. Type booléen

Comme en C, pour une question de rétrocompatibilité, tout ce qui est différent de 0 est VRAI et tout ce qui est égal à 0 est FAUX. Cependant, en C++ le type ***bool*** est une alternative pour les expressions conditionnelles. Il peut prendre deux valeurs ***true*** et ***false***.

1.5. Surcharge ou polymorphisme de traitement

En C++, des fonctions distinctes peuvent avoir le même nom, si elles possèdent des paramètres relativement différents par leur type ou par leur nombre. Cela permet d'en simplifier l'utilisation, en appelant la même fonction du point de vue de l'utilisateur, on peut lui passer des paramètres différents

Exemple de surcharge

```
int   CalculerPuissance(int, int);
float CalculerPuissance(float, int);
float CalculerPuissance(float, double);
```

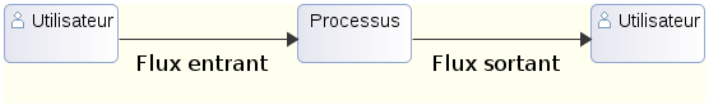
On parle ici de ***polymorphisme de traitement***, c'est le compilateur qui choisit la méthode à utiliser en fonction du type et du nombre de paramètres au moment de l'appel.

Cette surcharge peut également s'appliquer aux opérateurs. Classiquement, un opérateur peut agir sur deux variables de type simple, on pourra surcharger l'opérateur pour qu'il s'applique à de nouveaux types comme les classes pour obtenir un comportement similaire.

1.6. Entrées / Sorties : les flux

1.6.1. Généralités

Tout comme le langage C, le C++ ne dispose pas de mots clés spécifiques pour réaliser les entrées / sorties avec l'utilisateur ou plus généralement avec le système. Il dispose de librairies standards qui permettent de mettre en œuvre cela. En C++, les entrées / sorties sont réalisées à l'aide de mécanismes nommés **flux**. Un flux ou *stream* en anglais est une manière abstraite de représenter un flot de données entre un producteur d'information et un consommateur.



Le flux est toujours considéré du point de vue du processeur. C'est le processus qui reçoit l'information par le flux entrant et qui l'envoie à travers le flux sortant.

Les entrées et sorties utilisant les flux sont définies par deux classes déclarées dans le fichier d'en-tête ***<iostream>*** :

- ***ostream*** pour *Output stream*, définit le flux sortant. Cette classe surcharge l'opérateur d'insertion **<<**.
- ***istream*** pour *Input stream*, définit le flux entrant. Cette classe surcharge l'opérateur d'extraction **>>**.

1.6.2. Flux standards

Tout comme le langage C utilise l'entrée standard ***stdin*** et la sortie standard ***stdout***, le C++ associe un flux sortant vers cette sortie et un flux entrant depuis cette entrée. On trouve également un flux vers la sortie d'erreur et un dernier vers la sortie technique ou sortie d'information.

- ***cout*** : écrit vers la sortie standard, l'écran,
- ***cin*** : lit à partir de l'entrée standard, le clavier,
- ***cerr*** : écrit sur la sortie d'erreur, cette sortie est non tamponnée,
- ***clog*** : écrit sur la sortie technique, cette sortie est tamponnée.

Ces quatre instances représentent les flux standards en C++. Elles sont définies dans l'espace de nommage ***std***. Les deux dernières instances sont utilisées pour afficher ou mémoriser les messages d'erreur ou d'information. L'instance ***std::cerr*** ne possède pas de tampon cela implique que le message d'erreur s'affiche directement sans attendre que le tampon de sortie soit plein ou l'utilisation des manipulateurs ***std::flush*** ou ***std::endl***.

Le manipulateur ***std::endl*** écrit une fin de ligne **'\n'** sur le flux sortant puis vide le tampon. Le manipulateur ***std::flush*** vide uniquement le tampon d'écriture, le flux est physiquement envoyé sur l'unité de sortie.

Écriture sur la sortie standard ***bienvenue.cpp***

```
#include <iostream>          // pour cout
using namespace std;  // évite d'écrire std::cout

int main()
{
   cout << "Bienvenue en C++" ; // idem printf("Bienvenue en C++"); du langage C
   return 0;
}
```

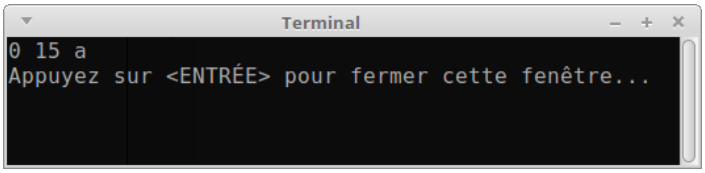
Écriture :

La classe ***ostream*** permet de réaliser un affichage formaté à la manière du ***printf*** de la librairie ***stdio*** en langage C. Ce formatage est beaucoup plus simple puisqu'il suffit d'enchaîner les opérateurs **<<,** comme le montre l'exemple ci-dessous. La surcharge de l'opérateur a été réalisée pour tous les types simples du C++ et les chaînes de caractères.

Écriture formatée ***formatage.cpp***

```
#include <iostream> // pour cout

using namespace std;        // évite d'écrire std::cout

int main()
{
      bool sortie = false;
      int  entier = 15;
      char car = 'a';
      cout << sortie << " " << entier << " " << car  << endl;
      return 0;
}
```

Le résultat obtenu est présenté à la figure suivante :



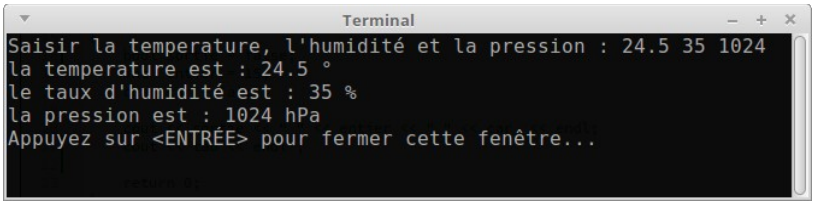
Lecture :

La classe ***istream*** permet la saisie de données à la manière du ***scanf*** de la librairie ***stdio*** du langage C. Pour cette classe, l'opérateur **>>** a été surchargé pour l'ensemble des types simples et les chaînes de caractères.

Lecture ***lecture.cpp***

```
#include <iostream>     // pour cin et cout
using namespace std;

int main()
{
    float temperature;
    float humidite;
    int pression;

    cout << "Saisir la température, l'humidité et la pression : " ;
    cin >> temperature >> humidite >> pression ;

    cout  << "la température est : " << temperature << " °"  << endl;
    cout  << "le taux d'humidité est : " << humidite << " %" << endl;
    cout  << "la pression est : " << pression <<  " hPa" << endl ;

    return 0;
}
```

La saisie des différentes données doit être séparée par un espace, une tabulation, ou un retour chariot. La dernière saisie est prise en compte après un retour chariot [Entrée].

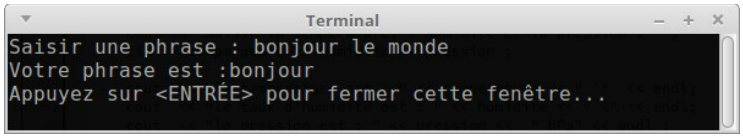


Cas particulier de la lecture des chaînes de caractères :

L'instance ***cin*** permet également la saisie de chaînes de caractères. Comme le montre l'exemple suivant, une difficulté se pose lorsque la chaîne est composée de plusieurs mots.

Lecture chaîne de caractères ***lecturemot.cpp***

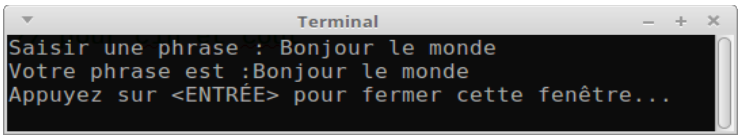
```
#include <iostream>     // pour cin et cout
using namespace std;

int main()
{
    char phrase[80+1];
    cout << "Saisir une phrase : " ;
    cin >> phrase ;
    cout  << "Votre phrase est :" << phrase << endl;

    return 0;
}
```



Seul le premier mot est mémorisé dans la variable, l'espace étant un caractère séparateur. La solution passe par l'appel de la méthode ***getline*** de la classe ***istream***. Le deuxième paramètre tient compte de la longueur maximale de la chaîne pour éviter tout débordement.

Lecture chaîne de caractères avec la prise en compte des espaces ***lecturechaine.cpp***

```
#include <iostream>     // pour cin et cout
using namespace std;

int main()
{
    char phrase[80+1];

    cout << "Saisir une phrase : " ;
    cin.getline(phrase,80) ;

    cout  << "Votre phrase est :" << phrase << endl;

 return 0;
}
```



1.6.3. Les manipulateurs de flux d'entrée-sortie

Ces manipulateurs sont des objets dont l'insertion dans un flux en modifie le fonctionnement. Comme le montre l'exemple suivant, l'affichage de la valeur peut s'effectuer en décimal, ou en hexadécimal.

Exemple de manipulateurs ***manipulateurs.cpp***

```
#include <iostream>
using namespace std;

int main()
{
    int val = 192;

    cout  << "Affichage par défaut     : " << val << endl;
    cout  << "Affichage en hexadécimal : " << hex << val << endl;
    cout  << "Affichage en décimal     : " << dec << val << endl;

    return 0;
}
```

Il existe deux types de manipulateurs, avec et sans argument.

Sans argument applicable à istream et ostream :

Rôle	Manipulateur	Valeur par défaut
Affichage symbolique des booléens	boolalpha	
	noboolalpha	X
Affichage de la base autre que la base 10	showbase	
	noshowbase	X
Affichage du point décimal	showpoint	
	noshowpoint	X
Affichage du signe + devant les nombres positifs	showpos	
	noshowpos	X
Affichage des caractères hexadécimaux en majuscule	uppercase	
	nouppercase	X
Affichage des nombres dans une base des bases : décimale, hexadécimale et octale (voir setbase)	dec	X
	hex	
	oct	
Mise en forme de l'affichage : aligné à gauche ou à droite	left	
	right	
Notation des nombres flottants	fixed	
	scientific	

Remarque

Lorsqu'un manipulateur a été envoyé sur le flux de sortie, le fonctionnement du flux reste inchangé jusqu'à la rencontre d'un prochain manipulateur qui contredit le premier. Par exemple, si l'affichage des nombres est demandé en hexadécimal avec le manipulateur hex, tous les nombres s'afficheront par la suite en hexadécimal jusqu'à l'utilisation du manipulateur dec ou oct.

Sans argument applicable à ostream :

Rôle	Manipulateur
Écriture de la fin de ligne '\n' sur le flux de sortie et vidage du tampon	endl
Écriture de la fin de chaîne '\0' sur le flux de sortie	ends
Vidage du tampon de sortie	flush

Sans argument applicable à istream :

Rôle	Manipulateur
Ignore les espaces précédant la lecture de données	ws
	skipws
Impose de ne pas saisir des espaces devant des données	noskipws

Manipulateur avec paramètres :

Ces manipulateurs demandent l'inclusion du fichier ***<iomanip>***. Ils sont utilisés principalement avec la classe ***ostream***.

Rôle	Manipulateur	Paramètre
Spécifier la largeur d'une zone	setw(int n)	Nombre de caractères
Spécification du caractère de remplissage	setfill(char c)	Caractères de remplissage, par défaut espace
Spécification de la base	setbase(int base)	La base est 8, 10, 16 équivalent à oct, dec, hex.
Spécification du nombre de chiffres significatifs	setprecision(int n)	Nombre de chiffres après la virgule

Exemple

Exemple de manipulateurs ***manipulateurs2.cpp***

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int val = 192;
    float val2 = 3.141592654;
    bool sortie = true;
    cout  << "Affichage par défaut     : " << val << endl;
    cout  << "Affichage en hexadécimal : " << hex << val << endl;
    cout  << "Affichage en décimal     : " << dec << val << endl;
    cout << hex << val << " " << uppercase << val << " " ;
    cout << showbase  << val << nouppercase << " " << val << endl;
    cout << "+" << setfill('-') << setw(21) << "+" << setfill(' ') << endl;
    cout << "|" << setw(20) << left << "abc" << "|" << endl;
    cout << "|" << setw(20) << right << "abc" << "|" << endl;
    cout << "+" << setfill('-') << setw(21) << "+" << setfill(' ') << endl;
    cout << dec << sortie << " " << boolalpha << sortie << endl;
    cout << val2 << " " << fixed << val2 << " " << scientific << val2 << " ";
    cout << fixed << setprecision(2) << val2 << endl;
    cout << "Entrez un nombre en octal : " ;
    cin >> oct >> val ;
    cout << "vous avez saisie " << dec << val << " en décimal" << endl;
    return 0;
}
```

```
Terminal
Affichage par défaut     : 192
Affichage en hexadécimal : c0
Affichage en décimal     : 192
c0 C0  0XC0 0xc0
+--------------------+
|abc                 |
|                 abc|
+--------------------+
1 true
3.14159 3.141593 3.141593e+00 3.14
Entrez un nombre en octal : 770
vous avez saisie 504 en décimal
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

1.6.4. Les flux basés sur des fichiers

Par analogie, le C++ dispose d'une librairie ***<fstream>*** qui définit deux objets ***ifstream*** et ***ofstream*** permettant de connecter un flux à un fichier texte par défaut. Ce sont des entités du type ***FILE*** rencontrées en langage C avec la bibliothèque ***stdio***. Le premier désigne un flux en provenance d'un fichier, le second désigne un flux vers un fichier. L'objet ***fstream*** est également présent dans cette librairie, il est d'usage général et permet de faire toutes sortes d'entrées, sorties sur un fichier, quel que soit le type de données manipulées. Dans ce cas, il est nécessaire de préciser le mode d'ouverture du fichier ce qui est implicite avec les deux premiers.

Le constructeur de ***ifstream*** et celui de ***ofstream*** reçoivent en paramètre d'entrée le nom d'un fichier texte à ouvrir. Pour le premier objet, l'ouverture se fait en lecture et pour le second en écriture. Comme pour chaque appel système, il est nécessaire de vérifier la bonne ouverture du fichier avant de poursuivre les traitements. Dans le cas d'un fichier en lecture, cela permet de vérifier que le fichier existe bien et que le programme à bien le droit de lire ce fichier. Pour un fichier ouvert en écriture, cela permet de s'assurer que le répertoire où il va être stocké est accessible en écriture ou que le disque n'est pas rempli.

À noter également, lorsqu'un fichier est ouvert en écriture par défaut, si celui-ci existe déjà sur le disque, il est écrasé à moins de le préciser dans un second paramètre.

Présentation des différents modes d'ouverture contenus dans la classe ***fstream*** :

Les principaux modes d'ouverture	
app	(**app**end) ouvre le flux en ajout, la prochaine écriture se fera à la fin du fichier. Celui-ci n'est pas détruit.
ate	(**at e**nd) Définit l'indicateur de position de flux à la fin du fichier lors de l'ouverture, utilisé pour la lecture.
binary	Le flux donne accès à un fichier de type binaire plutôt que de type texte.
in	(**in**put) Ouverture du flux en lecture, mode par défaut pour **ifstream**.
out	(**out**put) Ouverture du flux en écriture, mode par défaut pour **ofstream**.

Utilisation de la valeur par défaut, le mode n'est pas précisé. Il est défini par le nom de la classe.

Exemple d'ouverture de fichier texte en lecture ***ouvrefichier.cpp***

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    ifstream fichier("config.txt");
    if (!fichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;
    else
    {
        // traitement sur le fichier ouvert en écriture
    }
    return 0 ;
}
```

Les drapeaux peuvent être spécifiés et combinés avec l'opérateur OU binaire : |.

Exemple d'ouverture de fichier texte en lecture et écriture ***fichierdrapeauxcombines.cpp***

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    fstream fichier("exemple.txt", fstream::in | fstream::out);
    if (!fichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;
    else
    {
        // traitement sur le fichier ouvert en lecture et écriture
    }
    return 0 ;
}
```

Chaque lecture dans un fichier demande vérification pour être certain que la lecture a bien été effectuée. En effet à la fin du fichier, la ou les variables recevant la donnée ne sont pas forcément affectées. De même pour tester si la fin de fichier est atteinte il est nécessaire de faire au moins une lecture. Typiquement, la lecture d'un fichier s'effectue de la manière suivante :

Exemple de traitement typique de lecture de fichier *traitementfichier.cpp*

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string motCle ;
    // string remplace un tableau de caractères. la taille de la chaîne est dynamique
    int valeur;
    ifstream fichier("config.txt");
    if (!fichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;
    else
    {
        do
        {
            // le fichier contient sur chaque ligne des couples mot clé + valeur
            fichier >> motCle >> valeur ;
            if (fichier.good())//Si les valeurs ont bien été lues
            {
                // traitement des variables motCle et valeur
            }
        } while(!fichier.eof());
    }
    return 0 ;
}
```

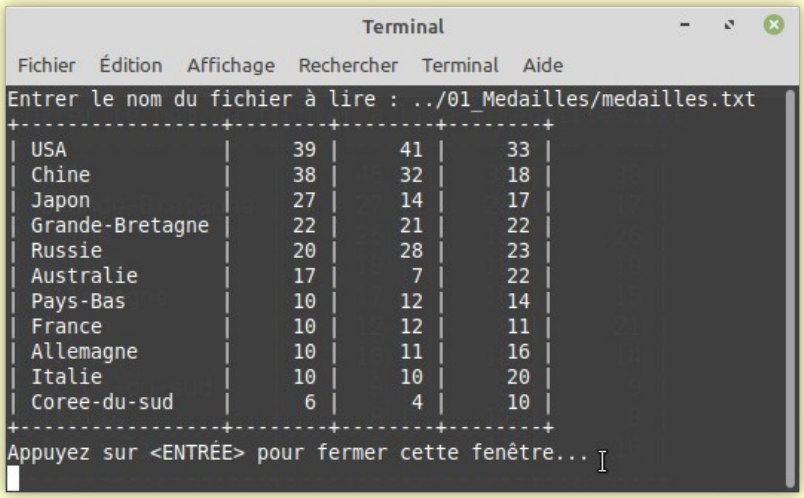
Exercice d'application

a) À partir du fichier texte, « medailles.txt », on souhaite obtenir le résultat :

medailles.txt

```
USA 39 41 33
Chine 38 32 18
Japon 27 14 17
Grande Bretagne 22 21 22
Russie 20 28 23
Australie 17 7 22
Pays-Bas 10 12 14
France 10 12 11
Allemagne 10 11 16
Italie 10 10 20
Corée-du-sud 6 4 10
```

Résultat attendu



b) On propose ici un extrait de code à compléter.

Exercice d'application ***medailles.cpp***

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>

using namespace std;


int main() {
    string nomDuFichier;

    cout << "Entrer le nom du fichier à lire : ";
    cin >> nomDuFichier;

    //Création du flux en lecture sur le fichier
    ifstream leFichier(nomDuFichier.c_str()); // c_str() transforme string en char*

    if (!leFichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;

    else
    {
        string pays;
        int nbOr;
        int nbArgent;
        int nbBronze;

        // A compléter, affichage de la première ligne du tableau
        do
        {
               //récupération des valeurs
            leFichier >> pays >> nbOr >> nbArgent >> nbBronze;

            if (leFichier.good())//Si les valeurs ont bien été lues
            {
                // A compléter, affichage de chaque ligne du tableau
            }
        } while (!leFichier.eof());
        // A compléter, affichage de la dernière ligne du tableau.
    }
    return 0;
}
```

Remarques

Le fichier **medailles.txt** doit se trouver dans le même répertoire que l'exécutable. Pour les autres cas, il est nécessaire de préciser le chemin complet.

c) Vous pouvez ne pas obtenir le bon résultat notamment à partir de l'affichage des informations pour la Grande-Bretagne ou la Corée-du-Sud, expliquer le problème, et modifier votre fichier texte en conséquence.

d) Écrire un programme qui produit le même résultat, mais dans un fichier texte au lieu de l'écran. Rappel, pour un fichier en écriture il est nécessaire d'inclure la librairie **fstream**.

1.7. Références

Le C++ introduit une nouvelle notion, il est possible d'utiliser la référence d'une variable ou d'une instance. Cette référence peut être assimilée à un alias, un autre nom pour désigner un même élément. Une référence ne peut être initialisée qu'une seule fois, lors de sa création. Toute autre modification affecte la variable référencée.

Exemple simple : déclaration, utilisation *reference.cpp*

```
#include <iostream>
using namespace std;

int main()
{
    int val1 = 124;
    int val2 = 0;
    int resultat ;

    int &ref1 = val1; //affectation de la référence ref1 à la variable val1
    resultat = ref1 + 5;  // équivaut à :  resultat = val1 + 5;

    int &ref2 = val2 ; //affectation de la référence ref2 à la variable val2
    ref2 = val1 + 10; // équivaut à :  val2 = val1 + 10;

    cout << "La variable resultat vaut : " << resultat << endl;
    cout << "La variable val2 vaut     : " << val2 << endl;

    return 0;
}
```

```
Terminal
La variable resultat vaut : 129
La variable val2 vaut     : 134
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

La déclaration se fait en faisant précédé le nom par le symbole **&**. L'utilisation se fait comme une variable simple, il n'y a pas d'étoile comme avec les pointeurs pour désigner le contenu ou plus précisément l'objet pointé.

Une référence peut être considérée comme un pointeur constant, elle ne peut pas être réaffectée ou incrémentée. L'utilisation principale de ces références est le passage de paramètres aux fonctions. Le passage de paramètres par références permet l'échange de données entre fonctions **en entrée** et **en sortie** comme avec les pointeurs tout en gardant la simplicité de l'utilisation de paramètres par valeur.

Exercice d'application

Afin d'étudier le passage de paramètres dans les trois cas de figure, par valeur, par adresse et par référence, le programme suivant est proposé :

Exemple simple : déclaration, utilisation *appelsdefonctions.cpp*

```
#include <iostream>
using namespace std;

int main()
{
    int a = 1;
    int b = -2;
    int c = 0;
    cout << "avant l'appel de Ajouter" << endl;
    cout << "a= " << a << endl;
    cout << "b= " << b << endl;
    cout << "c= " << c << endl;

    // Appel le la fonction Ajouter à compléter dans le tableau suivant le cas.

    cout << "après l'appel de Ajouter" << endl;
    cout << "a= " << a << endl;
    cout << "b= " << b << endl;
    cout << "c= " << c << endl;

    return 0;
}
```

1. À partir du programme précédent, on vous demande d'anticiper le résultat et de compléter l'appel de la fonction dans chacun des cas.

Fonctions	Appel de la fonction dans le programme ci-après	Avant	Après

void Ajouter(int a, int b ,int c)
{
c= a+b ;
}
		a=
		b=
		c=
			a=
			b=
			c=

void Ajouter(int a, int b ,int *c)
{
*c= a+b ;
}
		a=
		b=
		c=
			a=
			b=
			c=

void Ajouter(int a, int b ,int &c)
{
c= a+b ;
}
		a=
		b=
		c=
			a=
			b=
			c=

2. Comment se nomme chacun de ces trois passages de paramètres pour la variable c ?

De par sa simplicité d'utilisation, le passage de paramètre par référence est à privilégier en C++. Le nombre d'octets transmis lors de l'appel est comparable à celui transmis avec un pointeur, mais l'utilisation s'apparente à une simple variable passée par valeur.

Référence constante

Le compilateur C++ introduit la notion de paramètres constants, ***const***, qui permet d'avoir des paramètres uniquement **en entrée**. Ainsi, lorsqu'un paramètre transmis par référence est précédé du mot clé **const** à une fonction, celle-ci ne peut pas affecter de valeur à cette variable. Elle doit être utilisée uniquement en lecture, le compilateur se charge de cette vérification.

Exemple de fonction avec un paramètre constant

```
void Afficher(const string &chaine);
```

La fonction Afficher ne peut pas modifier la variable **chaine**, le mot clé **const** indique que le paramètre est uniquement en entrée pour cette fonction, bien qu'il soit passé par référence.

1.8. Constantes en C++

La notion de paramètre constant peut être généralisée à toutes les variables du C++. En effet, le C++ possède de véritables constantes, contrairement au langage C qui utilise le préprocesseur pour les définir.

Exemple de définition de constantes

```
#define NB_ELEMENTS 10  // Constante en C Attention il n'y a pas de ;
const int NB_ELEMENTS = 10; // définition d'une constante en C++, la constante est typée.


int tampon[NB_ELEMENTS];
```

Dans le premier cas, le préprocesseur remplace la chaîne de caractères par la valeur qui suit. Dans le cas du C++, la constante est véritablement typée, c'est bien ici un entier. Cette programmation est plus rigoureuse et, à privilégier en C++.

La réflexion peut être complétée par l'utilisation du mot clé ***const*** avec un pointeur, suivant la déclaration cela ne signifie pas la même chose en fonction de l'objectif voulu. Le souci est de toujours protéger les éléments pour plus de sécurité dans le code.

Déclaration	Signification	(ptr++)	((*ptr)++)
const char *ptr ;	ptr est un pointeur sur un caractère constant.	oui	non
int const *ptr ;	ptr est un pointeur constant sur un entier.	non	oui
const int * const ptr ;	ptr est pointeur constant sur un entier constant	non	non

1.9. Allocation et restitution de la mémoire

Le C++ introduit deux nouveaux opérateurs pour la gestion dynamique de la mémoire en remplacement des fonctions ***malloc***, ***calloc***, ***realloc*** et ***free*** de la librairie standard du langage C. Il s'agit de l'opérateur ***new*** pour allouer de la mémoire et de l'opérateur ***delete*** pour la restituer.

L'opérateur s'utilise de deux manières en fonction de la nature de l'objet pour lequel on souhaite allouer cette mémoire.

- Allocation pour un objet unique : ***new*** *type* ;
- Allocation pour un tableau d'objet : ***new*** *type*[n] ;

Dans les deux cas, l'opérateur ***new*** retourne un pointeur sur un objet *type*. Le C++ offre l'avantage de pouvoir allouer dynamiquement de la mémoire pour un tableau dont la taille n'est pas connue avant la compilation, contrairement au langage C.

Exemples d'allocation

```
int *ptr;       // déclaration du pointeur
ptr = new int ; // allocation mémoire pour un entier

int *tab;
tab = new int [10]; // allocation mémoire pour un tableau de 10 entiers
```

La restitution de la mémoire dynamique est réalisée par l'opérateur ***delete*** dans le premier cas et ***delete [ ]*** dans le second. En effet pour ce dernier cas, l'opérateur ***delete [ ]*** libère chacune des cases du tableau. Il est impératif d'utiliser la forme appropriée pour libérer la mémoire, sous peine de résultats imprévisibles.

Exemples de restitution

```
delete ptr;
delete [] tab; // tab a été alloué avec l'opérateur new int []
```

1.10. Traitement des erreurs : Les exceptions

1.10.1. Introduction

Un programme, même testé et réputé sans erreur, peut être amené à s'arrêter dans des circonstances autres que celles prévues par le programmeur. Il existe différentes sources d'erreurs :

- les erreurs détectées à la compilation, elles doivent être corrigées avant l'exécution
- les erreurs détectées et corrigées à l'exécution,
- les erreurs détectées et non traitées à l'exécution,
- les erreurs non détectées à l'exécution.

L'origine de ces erreurs peut-être multiple :

- les erreurs d'entrée/sortie, l'utilisateur fait une faute dans la saisie d'une donnée, un fichier est corrompu ou inaccessible,
- les erreurs matérielles, l'imprimante est déconnectée, le disque est saturé, la connexion réseau est défaillante,
- les erreurs de programmation, l'indice d'un tableau est invalide, emploie d'une variable non initialisée.

1.10.2. Traitement des erreurs

En présence d'une erreur, un programme se comporte selon trois cas de figure :

- **Idéalement** : il permet de revenir à un état défini et permet de poursuivre ainsi l'exécution.
- **Honnêtement** : il avertit l'utilisateur et permet une sortie correcte après une sauvegarde de l'ensemble du travail en cours.
- **Lamentablement** : il s'interrompt brusquement et affiche au mieux un message d'erreur peu clair.

Une solution consiste à vérifier à l'aide de structure de contrôle tous les cas de dysfonctionnement possibles. Toutefois, cette solution alourdit considérablement le code en cas de contrôles successifs. En langage C, le traitement d'erreur respecte certaines conventions de programmations :

- des valeurs de retours spéciales, un programme qui se finit bien retourne 0 ou plutôt **EXIT_SUCCESS** sinon, il retourne **EXIT_FAILURE** deux constantes définies dans **<sdtlib.h>**.
- Le positionnement de drapeaux, par exemple la variable **errno** définie dans **<errno.h>** contient un code d'erreur indiquant la nature de l'erreur.

Exemples de traitement d'erreur en C *erreur.c*

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int x;
    int y;
    int sortie = EXIT_SUCCESS;

    printf("Entrez deux nombres : ");
    if (scanf("%d %d", &x, &y) != 2)
    {
        printf("Vous deviez saisir deux nombres !\n");
        sortie = EXIT_FAILURE;
    }
    else
    {
        printf("Vous avez saisi : %d et %d\n", x, y);
    }
    return sortie;
}
```

Qui vérifie systématiquement que la saisie s'est bien déroulée ?
Il n'y a aucune garantie que les conventions soient suivies par tout le monde

En C++, un autre mécanisme a été mis en place, la gestion des exceptions. Si un problème survient, une exception est levée. Toute exception peut et doit être attrapée dans le bloc ***try***. Le traitement des exceptions est placé en dehors de la séquence normale, dans le bloc ***catch***. Un premier exemple montre la structure ***try … catch*** que nous détaillerons de manière plus approfondie dans le prochain chapitre sur les classes.

Exemples de traitement avec les exceptions *erreur.cpp*

```c
#include <iostream>
using namespace std;

int main()
{
    int nombre1, nombre2 ;
    try
    {
        cout << "Entrez deux nombres entiers : ";
        cin >> nombre1 >> nombre2 ;
        if(nombre2 == 0)
            throw string("Erreur de division par ZÉRO !");
        else
            cout << nombre1 / nombre2 << endl;
    }

    catch(string const& chaine)
    {
        cerr << chaine << endl;
    }
    return 0;
}
```

1.11. Conversion de types

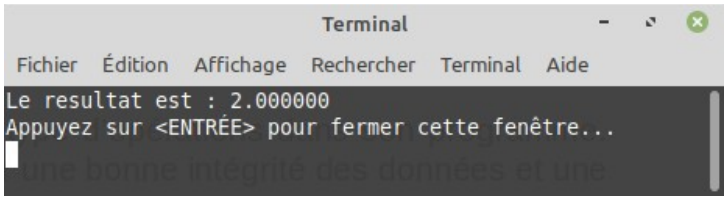
Le langage C++ tout comme le langage C à son époque sont des langages avec un **typage fort**. Il impose des restrictions sur la façon dont les différents types de variables peuvent être convertis les uns aux autres sans aucune instruction de conversion. Le compilateur interdit tout **transtypage**, autre terme utilisé, pour des conversions dépourvues de sens et avertit en cas de perte de précision ou risque de mauvaise interprétation de l'information.

Un programmeur peut cependant être amené à utiliser explicitement ce type d'opérations dans son programme. Dans ce cas, il faut le faire à bon escient et ne pas en abuser pour garantir une bonne intégrité des données et une meilleure maintenabilité du code.

Dans l'exemple suivant, on souhaite que le résultat de la division de deux entiers donne un résultat sous la forme d'un réel.

Exemple en langage C

```c
#include <stdio.h>
int main()
{
    int val1 = 9;
    int val2 = 4;
    float resultat;
    resultat = val1/val2;
    printf("Le resultat est : %f\n",resultat);
    return 0;
}
```



Sans opérateur de conversion, le résultat n'est pas celui attendu.
Avec l'opérateur de conversion **(<type>) <expression>** le résultat est tout autre

Exemple en langage C avec l'opérateur de conversion

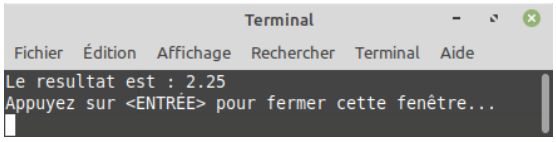
```c
#include <stdio.h>
int main()
{
    int val1 = 9;
    int val2 = 4;
    float resultat;
    resultat = val1/(float)val2;
    printf("Le resultat est : %f\n",resultat);
    return 0;
}
```

Le langage C++ distingue différentes situations de transtypage et utilise pour cela une syntaxe différente du langage C.

Le cas présent correspond au cas le plus simple, son rôle est d'expliciter les conversions implicites. L'opérateur est dans ce cas **static_cast<type> (expression)**. La conversion de type statique est toujours réalisée sans vérification. L'usage en est donc fait en toute conscience.

Exemple en langage C++ avec l'opérateur de conversion static_cast<>

```c
#include <iostream>
using namespace std;
int main()
{
    int val1 = 9;
    int val2 = 4;
    float resultat;
    resultat = val1/static_cast<float>(val2);
    cout << "Le resultat est : " << resultat << endl;
    return 0;
}
```



D'autres cas de conversion, impliquant une autre syntaxe, seront décrits dans ce document lorsque de nouvelles notions auront été abordées.

2. Définition de classes

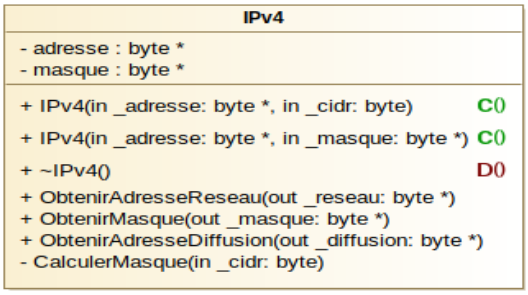
2.1. Introduction

La programmation-objet consiste dans un premier temps à spécifier les différentes classes de l'application. D'une manière générale, un fichier **.h** dit « d'en-tête » contient la déclaration d'une classe, un deuxième fichier **.cpp** contient la définition des différentes méthodes.

2.2. Notion d'encapsulation

L'encapsulation représente l'idée de regrouper sous une même entité les attributs et les méthodes. Ce concept apporte une certaine intégrité aux **données membres** ou **attributs** qui ne peuvent être manipulés que par les **fonctions membres** de la classe ou **méthodes**. L'autre intérêt est la portabilité du code produit puisque les programmes et les données attachées sont encapsulés dans une même entité pour ne former qu'un tout. Cette vision de la programmation permet de modéliser plus facilement les objets du monde réel.

Comme exemple, on propose d'étudier la classe **IPv4**. Cette classe possède deux attributs privés, un signe **–** devant chaque attribut dans la représentation UML, les rend accessibles uniquement par les méthodes de la classe. Comme les adresses IPv4 peuvent être représentées de deux manières, elle dispose de deux constructeurs, le premier permet d'initialiser les attributs à partir d'un tableau d'octets contenant une adresse IP et de son CIDR, nombre de bits utilisés pour son masque associé, exemple : ''192.168.1.1 /24''. Le deuxième constructeur initialise les attributs sous la forme d'un tableau contenant l'adresse IP et un second tableau représentant le masque, soit par exemple : ''192.168.1.1'' et ''255.255.255.0''.



La classe possède un destructeur, l'utilisation de pointeurs parmi les attributs laisse entendre une allocation dynamique de la mémoire et donc une restitution de cette mémoire dans le destructeur.

Elle possède également une méthode privée permettant de calculer le masque à partir du CIDR, ici le /24. Il n'y a pas de raison pour que cette méthode soit accessible par un autre programme d'où le qualificateur **privé**, le signe **–** dans la représentation UML, devant la méthode **CalculerMasque**.

Par contre, il est possible d'obtenir l'adresse réseau, l'adresse de diffusion et éventuellement le masque du réseau d'une adresse IPv4. Ces trois dernières méthodes possèdent donc un qualificateur **public** correspondant au signe **+** d'UML. Elles sont donc visibles par tous les programmes qui utilisent cette classe.

Déclaration de la classe IPv4

Déclaration de la classe IPv4 : ***ipv4.h***

```
#ifndef _IPV4_H
#define _IPV4_H

class IPv4
{
  private:
    unsigned char * adresse;
    unsigned char * masque ;
    void CalculerMasque(unsigned char _cidr); // _cidr est utilisé en tant que variable locale dans la
                                              // méthode (pas de const devant)

  public:
    IPv4(const unsigned char * _adresse, const unsigned char _cidr);
    IPv4(const unsigned char * _adresse, const unsigned char * _masque);
    ~IPv4();
    void ObtenirMasque(unsigned char * _masque);
    void ObtenirAdresseReseau(unsigned char * _reseau);
    void ObtenirAdresseDiffusion(unsigned char * _diffusion);
};

#endif
```

Voici quelques règles de qualité logicielle pour une programmation maintenable, efficace et compréhensible.

Codage C++	Explications
#ifndef _IPV4_H #define _IPV4_H ... #endif	Protège contre les inclusions multiples, la plupart des outils ajoutent ces lignes automatiquement lors de la création d'une classe C++.
_IPV4_H	Une constante ou définition est écrite en lettres majuscules, éventuellement séparées par des _.
class IPv4	Le nom d'une classe commence par une majuscule.
private:	Les attributs sont privés et ne sont donc pas modifiables en dehors de la classe.
unsigned char * adresse;	Les attributs et les variables d'une manière générale commencent par une minuscule.
const unsigned char * _adresse	Un paramètre passé à une fonction commence par un _ et porte le nom de l'attribut qu'il va initialiser.
const	Ce qualificateur indique que le paramètre est uniquement en entrée.
unsigned char * _adresse	Aurait pu être remplacé par unsigned char _adresse[ ].
void **CalculerMasque**();	Le nom d'une méthode est un verbe à l'infinitif suivi d'un complément. La première lettre commence par une majuscule.
ObtenirAdresseReseau	Une majuscule sépare chaque mot différent dans un identifiant.

Toutes ces recommandations ne sont pas obligatoires, mais vivement recommandées. Un code, même écrit par plusieurs personnes, doit toujours suivre les mêmes règles de codage.

Implémentation de la classe IPv4

Cette première partie du code montre comment implémenter en C++ les deux constructeurs de la classe. Pour chacun d'eux, il y a une allocation mémoire pour l'adresse IP et le masque. Le destructeur restitue la mémoire allouée. La méthode privée **CalculerMasque** est uniquement appelée dans le premier constructeur afin d'initialiser l'attribut **masque**.

Codage des constructeurs, et du destructeur : ***ipv4.cpp***

```
#include "IPv4.h"

IPv4::IPv4(const unsigned char * _adresse,const unsigned char _cidr)
{
    adresse = new unsigned char [4];
    masque  = new unsigned char [4];
    for(int indice = 0 ; indice < 4 ; indice++)
        adresse[indice] = _adresse[indice];
    if(_cidr <= 32)
        CalculerMasque(_cidr);
}

IPv4::IPv4(const unsigned char * _adresse,const unsigned char * _masque)
{
    adresse = new unsigned char [4];
    masque  = new unsigned char [4];
    for(int indice = 0 ; indice < 4 ; indice++)
    {
        adresse[indice] = _adresse[indice];
        masque[indice]  = _masque[indice];
    }
}

IPv4::~IPv4()
{
    delete [] adresse;
    delete [] masque ;
}
```

Le détail de la fonction est donné ci-après :

Codage de la fonction CalculerMasque *ipv4.cpp*

```
void IPv4::CalculerMasque(unsigned char _cidr)
{
    int indice ;
    // Le masque est remis à 0 -> 0.0.0.0
    for(indice = 0 ; indice < 4 ; indice++)
        masque[indice] = 0 ;

    indice = 0;
    // tant que le cidr est un multiple de 8
    while(_cidr >= 8)
    {
        masque[indice++] = 255 ;
        _cidr -= 8 ;
    }

    // Complément pour la fin du cidr (<8)
    unsigned char puissance = 128 ;
    while(_cidr-- > 0) // Après le test la variable _cidr est décrémentée
    {   // les puissances de 2 sont ajoutées à l'octet par valeur décroissante
        masque[indice] += puissance ;
        puissance /=2 ;
    }
}
```

Pour la suite du code, chaque méthode **Obtenir…** possède un paramètre de sortie, le choix, ici, est d'utiliser un pointeur, chaque fonction reçoit le nom d'un tableau (donc un pointeur constant) déclaré dans le programme principal. Ainsi, la mise à jour est automatique dans le programme appelant.

La méthode **ObtenirMasque** ne fait que recopier le masque dans le tableau passé en paramètre.

Codage de la méthode ObtenirMasque : *ipv4.cpp*

```
void IPv4::ObtenirMasque(unsigned char * _masque)
{
    for(int indice = 0 ; indice < 4 ; indice++)
        _masque[indice] = masque[indice];
}
```

Pour obtenir une adresse réseau, on rappelle ici, qu'il s'agit de faire un **ET** binaire entre chaque bit de l'adresse IP et le masque de réseau. La fonction se code de la manière suivante :

Codage de la méthode ObtenirAdresseReseau : *ipv4.cpp*

```
void IPv4::ObtenirAdresseReseau(unsigned char * _reseau)
{
    for(int indice = 0 ; indice < 4 ; indice++)
        _reseau[indice] = adresse[indice] & masque[indice] ;
}
```

	192	168	1	1	Adresse IP
ET	1100 0000	1010 1000	0000 0001	0000 0001	
	255	255	255	0	Masque
	1111 1111	1111 1111	1111 1111	0000 0000	
soit	1100 0000	1010 1000	0000 0001	0000 0000	Adresse réseau
	192	168	1	0	

L'opération **ET** binaire est effectué bit à bit : 1 **ET** 1 = 1 sinon 0

Pour obtenir l'adresse de diffusion, il s'agit de compléter l'adresse réseau en plaçant le reste des bits à 1. On peut complémenter le masque du réseau et faire un **OU** binaire avec l'adresse réseau.

Codage de la méthode ObtenirAdresseDiffusion : ***ipv4.cpp***

```
void IPv4::ObtenirAdresseDiffusion(unsigned char *_diffusion)
{
    unsigned char adresseDuReseau[4];
    ObtenirAdresseReseau(adresseDuReseau);
    for(int indice = 0 ; indice < 4 ; indice++)
        _diffusion[indice] = adresseDuReseau[indice] | ~masque[indice] ;
}
```

	192	168	1	0	Adresse réseau
OU	1100 0000	1010 1000	0000 0001	0000 0000	
	0	0	0	255	$\overline{\text{Masque}}$
	0000 0000	0000 0000	0000 0000	1111 1111	
soit	1100 0000	1010 1000	0000 0001	1111 1111	Adresse diffusion
	192	168	1	255	

L'opération **OU** binaire est effectué bit à bit : 0 **OU** 0 = 0 sinon 1

Exercice d'application

1. La classe IPv4 peut être complétée par une méthode fournissant : l'adresse de la première machine du réseau, une autre fournissant celle de la dernière et éventuellement une troisième avec un paramètre de retour retournant le nombre de machines dans le réseau.

Vérification en ligne du résultat : http://cric.grenoble.cnrs.fr/Administrateurs/Outils/CalculMasque/

Pour tester cette classe, on propose le programme principal suivant :

Programme principal : ***reseau.cpp***

```
#include <iostream>
#include "IPv4.h"
using namespace std;

void AfficherTableau(const unsigned char *tab);

int main()
{
    unsigned char adresse[4]= {192,168,1,1};
    unsigned char masque[4];
    unsigned char reseau[4];
    unsigned char diffusion[4];

    IPv4 uneAdresse(adresse, 24); // instanciation de la classe IPv4

    cout << "Adresse IPv4 : ";
    AfficherTableau(adresse);
    uneAdresse.ObtenirMasque(masque); // appel d'une méthode
    cout << "Masque : ";
    AfficherTableau(masque);
    uneAdresse.ObtenirAdresseReseau(reseau);
    cout << "Réseau : ";
    AfficherTableau(reseau);
    uneAdresse.ObtenirAdresseDiffusion(diffusion);
    cout << "Diffusion : ";
    AfficherTableau(diffusion);

    return 0;
}
```

Cette fonction évite la répétition de code pour l'affichage

Fonction AfficherTableau *reseau.cpp*

```
void AfficherTableau(const unsigned char *tab)
{
    for(int indice=0 ; indice < 4 ; indice ++)
    {
        cout  << static_cast<int> (tab[indice]);
        if(indice < 3)
            cout << "." ;
    }
    cout << endl;
}
```

```
Terminal
Autre adresse : Adresse IPv4 : 192.168.1.1
Masque : 255.255.255.0
Réseau : 192.168.1.0
Diffusion : 192.168.1.255
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

2.3. Surcharge d'opérateurs

Le C++ permet de surcharger les opérateurs utilisés pour les types de base pour les classes qu'il est amené à définir afin de simplifier l'écriture. La seule contrainte est d'indiquer au compilateur comment réaliser l'opération.

Opérateur entre deux éléments de type de base *surcharge_base.cpp*

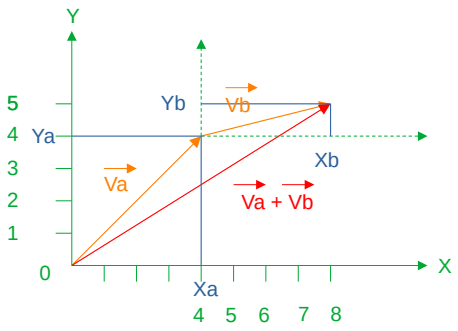
```
#include <iostream>

using namespace std;

int main()
{
    int x = 10;
    int y = 20;
    int resultat = x + y ;
    cout << "Le résultat de l'addition est : " << resultat << endl;
    return 0;
}
```

Dans ce premier exemple, l'addition entre deux entiers est connue par le compilateur. Dans le second exemple ci-après, si la classe Vecteur est définie, il devient nécessaire d'expliciter ce que représente l'addition de deux vecteurs.

Programme principal addition de vecteurs *main_vecteur.cpp*

```
#include <iostream>
#include "vecteur.h"
using namespace std;
int main()
{
    Vecteur va(4,4);
    Vecteur vb(4,1);
    Vecteur resultat;
    resultat = va + vb;
    resultat.Afficher();
    return 0;
}
```

```
Terminal
Fichier  Édition  Affichage  Recherche  Terminal  Aide
Vecteur : (8,5)
Press <RETURN> to close this window...
```



Pour réaliser cela, la classe vecteur est définie ainsi :

Opérateur entre deux instances d'une même classe *vecteur.h*

```
#ifndef VECTEUR_H
#define VECTEUR_H
class Vecteur
{
public:
    Vecteur(const int _x,const int _y);// ce premier constructeur initialise x et y avec les paramètres
    Vecteur(); // celui-ci initialise x et y à 0
    Vecteur operator+ (const Vecteur &_autre);
    void Afficher();
private:
    int x;
    int y;
};
#endif // VECTEUR_H
```

La surcharge de l'opérateur traduit l'opération à effectuer pour la classe Vecteur.

Surcharge de l'opérateur + *vecteur.cpp*

```
Vecteur Vecteur::operator+(const Vecteur &_autre)
{
    return Vecteur(x +_autre.x, y + _autre.y);
}
```

Remarque

Dans ce cas précis, l'accès aux membres privés de la classe est possible pour réaliser l'opération.

Les différents opérateurs numériques et logiques peuvent être surchargés au besoin. Seul les opérateurs suivant ne peuvent être surchargés :

: :	.	.*	?:	Sizeof
typeid	static_cast	dynamic_cast	const_cast	reinterpret_cast

Dans l'exemple précédent, la surcharge portait uniquement sur un opérateur numérique. Le test d'égalité entre deux vecteurs peut également être envisagé avec la surcharge de l'opérateur ==

Surcharge de l'opérateur == *vecteur.cpp*

```
bool Vecteur::operator==(const Vecteur &_autre)
{
    bool retour = false;
    if(x == _autre.x && y == _autre.y)
        retour = true;
    return retour;
}
```

La surcharge des opérateurs permet de simplifier l'écriture ainsi l'utilisation dans le programme principal peut être :

Exemple d'utilisation de la surcharge de l'opérateur == *main_vecteur.cpp*

```
#include <iostream>
#include "vecteur.h"
using namespace std;
int main()
{
    Vecteur va(4,4);
    Vecteur vb(4,1);

    if(va == vb)
        cout << "les vecteurs sont identiques" << endl;
    else
        cout << "va est différent de vb" << endl;
    return 0;
}
```

Exercice d'application :

Réalisez pour la classe **Vecteur** la surcharge des opérateurs +=, -, -=, *, *= pour ces deux derniers opérateurs vous coderez le produit vectoriel de deux vecteurs.

2.4. Accesseurs et mutateurs

Pour faciliter l'affichage ou la modification d'un attribut privé ou protégé, le programmeur peut proposer des fonctions membres remplissant ces rôles ainsi :

Un **accesseur** est la fonction membre permettant de récupérer le contenu d'un attribut avec une étiquette privée ou protégée.

Un **mutateur** est une fonction membre permettant la modification de l'attribut.

Dans la terminologie anglophone, il est question de **getter** et de **setter**, c'est pourquoi par convention, le nom de ces fonctions membres est composé du nom de l'attribut précéder du préfixe **get** pour l'accesseur et **set** pour le mutateur.

Déclaration des accesseurs et mutateurs *vecteur.h*

```
#ifndef VECTEUR_H
#define VECTEUR_H
class Vecteur
{
public:
    Vecteur(const int _x,const int _y); // ce premier constructeur initialise x et y avec les paramètres
    Vecteur(); // celui-ci initialise x et y à 0
    int  getX() const;        // accesseur pour l'attribut x
    void setX(const int _newX);      // mutateur pour l'attribut x
    Vecteur operator+ (const Vecteur &_autre);
    bool operator==(const Vecteur &_autre);
    void Afficher();
private:
    int x;
    int y;
};
#endif // VECTEUR_H
```

La définition des accesseurs et mutateurs dans le fichier sources est relativement simple :

Codage des accesseurs et mutateurs *vecteur.cpp*

```
int Vecteur::getX() const
{
    return x;
}
void Vecteur::setX(const int _newX)
{
    x = _newX;
}
```

Remarque

Les outils de développement évolués offre généralement une fonctionnalité automatique pour créer les accesseurs et mutateur à partir des attributs.

Exercice d'application

Réalisez l'accesseur et le mutateur pour l'attribut y.

2.5. Valeur par défaut à un paramètre d'une méthode

Par mesure de simplification, il est courant d'affecter une valeur par défaut à un paramètre. Ainsi si le paramètre est omis lors de l'appel de la méthode, il prendra la valeur prévue par défaut.

Dans le cas de notre exemple de vecteur on peut très bien envisager une valeur par défaut à 0 pour x et y. Dans ce cas le constructeur sans paramètre n'a plus d'utilité ici. La déclaration de la classe devient donc :

Valeur par défaut aux paramètres *vecteur.h*

```
#ifndef VECTEUR_H
#define VECTEUR_H
class Vecteur
{
public:
    Vecteur(const int _x=0,const int _y=0);
    int  getX() const;
    void setX(int _newX);
    int  getY() const;
    void setY(int _newY);
    Vecteur operator+ (const Vecteur &_autre);
    bool operator==(const Vecteur &_autre);
    void Afficher();
private:
    int x;
    int y;
};
#endif // VECTEUR_H
```

Remarque

Les valeurs par défaut sont présentes uniquement dans la déclaration de la fonction.

Codage du constructeur *vecteur.cpp*

```
Vecteur::Vecteur(const int _x, const int _y):
    x(_x),
    y(_y)
{
}
```

La définition du constructeur reste inchangé, il initialise toujours les attributs, avec les valeurs par défaut ou pas.

2.6. Fonctions amies

Le C++, introduit la notion de **fonction amie**, ou "**friend function**". Ce concept permet à certaines fonctions d'accéder directement aux membres privés et protégés d'une classe, ce qui est normalement interdit par les règles d'encapsulation de la programmation orientée objet. Dans certaines situations spécifiques, il peut être utile pour des raison de simplification d'écriture de permettre à une fonction ou une autre classe d'accéder directement aux membres internes d'une classe sans pour autant exposer ces membres au reste du programme.

2.6.1. Déclaration d'une fonction amie

Le mot-clé **friend** placé dans la déclaration de la classe devant une fonction indique que cette fonction n'est pas un membre de la classe, mais une fonction amie.

Valeur par défaut aux paramètres *vecteur.h*

```
#ifndef VECTEUR_H
#define VECTEUR_H
class Vecteur
{
public:
    Vecteur(const int _x=0,const int _y=0);
    int  getX() const;
    void setX(int _newX);
    int  getY() const;
    void setY(int _newY);
    Vecteur operator+ (const Vecteur &_autre);
    bool operator==(const Vecteur &_autre);
    void Afficher();
    friend ostream & operator << (ostream &_flux,const Vecteur &_autre);
private:
    int x;
    int y;
};
#endif // VECTEUR_H
```

Dans cet exemple la surcharge de l'opérateur **<<** permet de manière simple de rediriger vers un flux les deux attributs

2.6.2. Implémentation d'une fonction amie

Cette fonction peut-être implémentée dans le même fichier contenant la définition des méthodes de la classe. Attention pour autant, ce n'est pas un membre de la classe.

Codage de la fonction amie *vecteur.cpp*

```
ostream & operator << (ostream &_flux,const Vecteur &_autre)
{
    _flux << "(" <<_autre.x << "," << _autre.y << ")" << endl ;
}
```

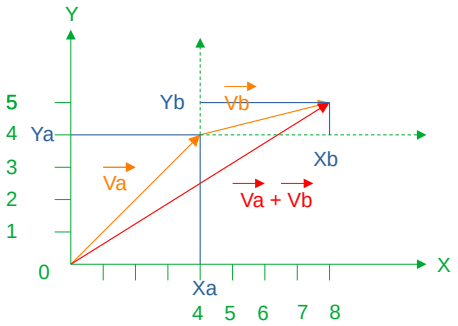
Les fonction amies peuvent être particulièrement utiles comme montre l'exemple précédent pour la surcharge de certains opérateurs.

L'appel de la fonction amie est réalisée tout simplement de la manière suivante :

Appel de la fonction amie **main.*cpp***

```
#include <iostream>
#include "vecteur.h"
using namespace std;
int main()
{
    Vecteur va(4,4);
    Vecteur vb(4,1);
    Vecteur resultat;
    resultat = va + vb;
    cout << "Vecteur : " << resultat << endl;
    return 0;
}
```

```
Terminal
Fichier  Édition  Affichage  Recherche  Terminal  Aide
Vecteur : (8,5)
Press <RETURN> to close this window...
```



A comparer avec l'exemple du paragraphe 2.3.

Remarque

Les fonctions amies offrent une flexibilité supplémentaire, leur utilisation peut cependant compromettre l'encapsulation, qui est un principe fondamental de la programmation orientée objet. Il est donc recommandé de les utiliser avec parcimonie.

2.7. Pointeur « this »

Le pointeur **`this`** est un pointeur implicite disponible dans le contexte des méthodes **non statiques** d'une classe. Il pointe vers l'objet pour lequel la méthode est appelée, permettant ainsi d'accéder aux membres (attributs et méthodes) de cet objet. C'est un pointeur constant qui représente l'adresse de l'instance de la classe en question.

Il est particulièrement utile lorsque l'on souhaite passer en paramètre un pointeur sur une instance d'une classe à un autre objet.

Remarque

- Il est toujours implicitement transmis à chaque appel de fonction membre non statique.
- Il n'est pas nécessaire de le déclarer explicitement.
- Il peut être utilisé pour accéder aux membres (variables et fonctions) de l'objet en cours.
- Il est de type pointeur vers l'instance de la classe.
- Il est constant par défaut, ce qui signifie qu'il n'est pas possible de modifier l'adresse de l'objet qu'il pointe.

2.8. Modèle canonique dit Coplien

La forme canonique, dite de **Coplien**, définit un cadre de base à respecter pour les classes non triviales dont certains attributs peuvent être alloués dynamiquement.

Pour répondre à ce critère, une classe possède une forme canonique ou forme normale ou encore forme standard, si elle présente les méthodes suivantes :

- Un constructeur par défaut,
- Un constructeur de recopie,
- Un destructeur éventuellement virtuel
- La surcharge de l'opérateur d'affectation

Dans notre exemple précédent, pour la classe **IPv4**, ce n'est pas le cas. Notre classe ne possède pas de constructeur par défaut c'est-à-dire sans paramètre. Elle ne possède pas non plus de constructeur de recopie, et l'opérateur d'affectation n'est pas surchargé. Elle possède bien un destructeur. Par rapport à notre exemple de programme principal, cela était sans importance, le résultat obtenu est bien celui attendu. Maintenant, pour des traitements particuliers cela peut s'avérer problématique.

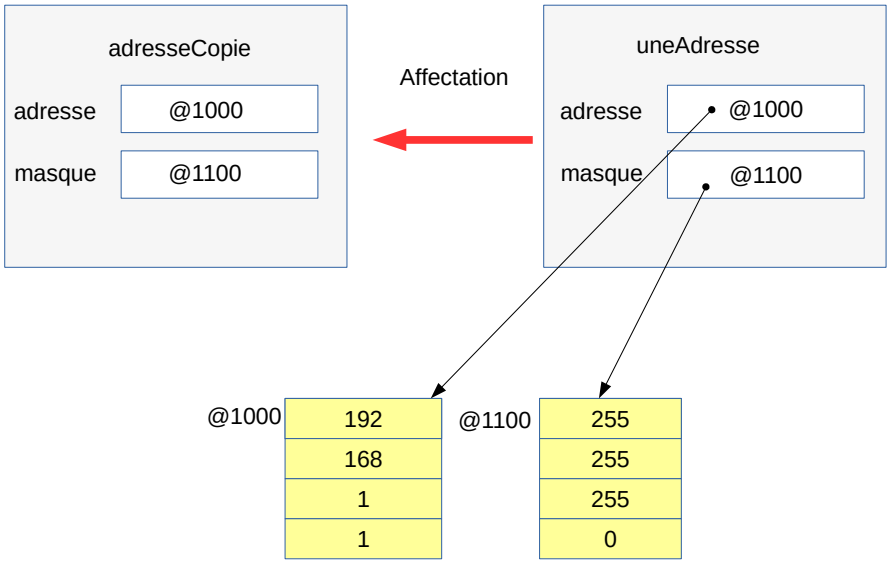
Pour illustrer ce propos, voici un nouveau programme principal, il déclare deux instances de la classe IPv4, la première est allouée dynamiquement, la seconde est une copie de la première. Ensuite, à partir de la copie, l'adresse réseau est affichée, avant et après destruction de l'instance de la première.

Nouveau programme principal : ***reseau2.cpp***

```
#include <iostream>
#include "IPv4.h"
using namespace std;

void AfficherTableau(const unsigned char *tab);

int main()
{
    unsigned char adresse[4]= {192,168,1,1};
    unsigned char reseau[4];

    IPv4 * uneAdresse = new IPv4(adresse, 24); // instanciation de la classe IPv4
    IPv4 adresseCopie = *uneAdresse; // l'instance est recopiée dans une autre

    cout << "Adresse réseau avant : ";
    adresseCopie.ObtenirAdresseReseau(reseau);
    AfficherTableau(reseau);

    delete uneAdresse; // destruction de la première instance

    cout << "Adresse réseau après destruction : ";
    adresseCopie.ObtenirAdresseReseau(reseau);
    AfficherTableau(reseau);

    return 0;
}
```

```
Terminal
Adresse réseau avant : 192.168.1.0
Adresse réseau après destruction : 0.0.0.0
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Constat fait, notre classe ne permet pas de réaliser cette opération. Lors de la copie, seuls les pointeurs ont été recopiés. La mémoire ayant été restituée lors de la destruction de la première instance, les pointeurs de la seconde pointent dans « le vide ».



Intérêt

La forme canonique dite de Coplien d'une classe impose la duplication de la mémoire allouée dynamiquement. Ainsi lorsqu'il y a une affectation ou un passage de paramètres par valeur, l'ensemble des données est bien dupliqué. Les deux instances possèdent une durée de vie dissociée.

Cette deuxième version présente la classe **IPv4** sous sa forme canonique dite de **Coplien**.

Déclaration de la classe IPv4 sous sa forme canonique : *ipv4_coplien.h*

```
#ifndef _IPV4_H
#define _IPV4_H

class IPv4
{
  private:
    unsigned char * adresse;
    unsigned char * masque ;

  public:
    IPv4(const unsigned char * _adresse, const unsigned char _cidr);
    IPv4(const unsigned char * _adresse, const unsigned char * _masque);
    ~IPv4(); // destructeur
    // Ajout pour la forme canonique en plus du destructeur
    IPv4(); // constructeur par défaut
    IPv4(const IPv4& _ipv4); // constructeur de copie
    IPv4 &operator= (const IPv4& _ipv4); // opérateur d'affectation
  private:
    void CalculerMasque(const unsigned char _cidr);
  public:
    void ObtenirMasque(unsigned char * _masque);
    void ObtenirAdresseReseau(unsigned char * _reseau);
    void ObtenirAdresseDiffusion(unsigned char * _diffusion);
};

#endif
```

Définition des nouveaux constructeurs, le premier alloue simplement de la mémoire, le second alloue la mémoire et recopie les données transmises en paramètre.

Définition des nouveaux constructeurs : *ipv4_coplien.cpp*

```
IPv4::IPv4()

{
    adresse = new unsigned char [4];
    masque  = new unsigned char [4];
}


IPv4::IPv4(const IPv4 &_ipv4)
{
    adresse = new unsigned char [4];
    masque  = new unsigned char [4];
    for(int indice = 0 ; indice < 4 ; indice++)
    {
        adresse[indice] = _ipv4.adresse[indice];
        masque[indice]  = _ipv4.masque[indice];
    }
}
```

Remarque

Dans le cas présent, bien que privés, les attributs de l'objet **_ipv4** sont accessibles dans le nouvel objet en construction.

Pour la surcharge de l'opérateur d'affectation, la mission est un peu plus délicate. En effet, il faut s'assurer :

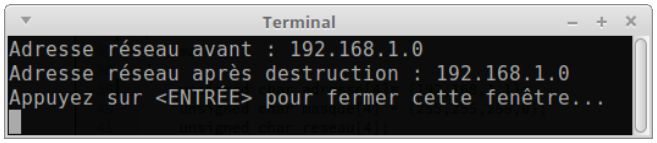
- que l'objet n'est pas lui-même, dans ce cas, il est retourné directement.
- ou, si la mémoire avait déjà été allouée dynamiquement pour ces variables, il faut la restituer avant d'affecter de nouvelles données.

Voici le code correspondant pour la classe **IPv4** :

Définition de l'opérateur d'affectation : *ipv4_coplien.cpp*

```
IPv4 &IPv4::operator=(const IPv4 &_ipv4)
{
    if(adresse != _ipv4.adresse || masque != _ipv4.masque)
    {
       if(adresse != nullptr && masque != nullptr)
       {
           delete [] adresse;
           delete [] masque ;
       }
       adresse = new unsigned char [4];
       masque  = new unsigned char [4];
       for(int indice = 0 ; indice < 4 ; indice++)
       {
           masque[indice]  = _ipv4.masque[indice];
           adresse[indice] = _ipv4.adresse[indice];
       }
    }
    return *this;
}
```

Le reste de la classe est inchangée. Voici le résultat :



Exercice d'application :

Décrivez les lignes de code suivant en indiquant quelle partie de la classe **IPv4** sous forme canonique a été mise en jeu :

Exercice

1. IPv4 adresseIP;
2. IPv4 adresse1(adresseIP);
3. IPv4 adresse2 = adresseIP;
4. adresse1 = adresseIP;

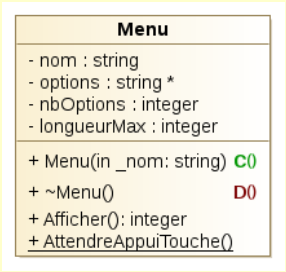
1. :
2. :
3. :
4. :

2.9. Application

Note au lecteur

Dans ce paragraphe, il est demandé de faire l'étude d'un menu pour sélectionner une option parmi plusieurs proposées. Cette application représente une synthèse des éléments vus jusqu'à maintenant. La classe **Menu**, que nous allons construire, sera réutilisée dans d'autres parties de ce cours.

L'objectif est d'afficher un menu chargé à partir d'un fichier et d'assurer son fonctionnement. Ce menu doit être parfaitement dynamique et s'adapter au fichier d'entrée.

menu.txt	Résultat attendu	Représentation UML
Option 1 Option 2 Option 3 Option 4 Quitter		

Description des Attributs	
nom	Désigne le nom du fichier
options	Représente un tableau de chaînes de caractères implémentées sous la forme de string. Ce tableau sera alloué dynamiquement en fonction du nombre de lignes du fichier.
nbOptions	Contient le nombre d'options du Menu
longueurMax	Taille de la plus grande chaîne contenue dans le tableau

La méthode **Menu::AttendreAppuiTouche()** est déclarée statique. Elle pourra être appelée même si la classe **Menu** n'est pas instanciée. Pour les autres méthodes, rien de particulier.

a) Déclarez la classe Menu dans un fichier menu.h

b) Complétez le code du constructeur à partir de l'algorithme suivant

Constructeur de la classe Menu *menu.cpp*

```
Menu::Menu(const string &_nom):nom(_nom), longueurMax(0)
{
        // ouvrir le fichier
        // Si il y a une erreur
        //      alors Afficher un message indiquant une erreur de lecture
        //              et mettre nbOptions à 0
        //      Sinon calculer nbOptions, le nombre d'options dans le fichier
        //              allouer dynamiquement le tableau options en fonction de nbOptions
        //              Pour chaque option dans le fichier
        //                      Lire l'option et l'affecter dans le tableau options
        //                      Si la taille de l'option est plus grande que longueurMax
        //                              alors longueurMax reçoit la taille de l'option
        //                      FinSi
        //              FinPour
        // FinSi
}
```

Quelques compléments d'information pour le codage de la méthode :

Pour calculer le nombre de lignes dans un fichier, on peut faire une première lecture et compter le nombre de lignes au fur et à mesure.

La ligne ci-dessous réalise le même traitement. Elle est à utiliser telle quelle, des explications supplémentaires interviendront à la fin du document dans les chapitres sur les **templates** et les algorithmes génériques.

```
nbLignes = static_cast<int>(count(istreambuf_iterator<char>(fichierMenu),istreambuf_iterator<char>(),'\n'));
```

Réalisation d'un transtypage, le résultat doit être un entier

Nom de la variable de type ifstream

La fonction **count** nécessite l'inclusion de la librairie Algorithm : #include <algorithm>

Il ne faut pas oublier de revenir au début du fichier pour reprendre la lecture de chaque ligne après.

```
fichierMenu.seekg(0,ios::beg);
```

http://www.cplusplus.com/reference/istream/istream/seekg/

La lecture de la ligne se fait par la fonction getline. La taille de la chaîne de caractères est donnée par la méthode length de la classe std::String. Ces éléments sont décrits dans la documentation de référence de cette classe. http://www.cplusplus.com/reference/string/string/.

c) Codez le destructeur, il y a eu une allocation dynamique dans le constructeur

d) Codez à présent la méthode **int Menu::Afficher()** en respectant la représentation donnée en exemple. Elle assure également la saisie du choix de l'utilisateur et le retourne. Si ce choix n'est pas dans les valeurs admises, l'affichage devient :

Pour la saisie, on ajoute à ***cin*** un test pour être bien sûr d'effectuer une saisie et supprimer les retours chariot superflus

```
if(!(cin>>choix))
{
  cin.clear();
  cin.ignore(std::numeric_limits<streamsize>::max(),'\n');
  choix = -1;
}
```

http://www.cplusplus.com/reference/ios/ios/clear/
http://www.cplusplus.com/reference/istream/istream/ignore/

Les deux lignes ***cin.clear()*** et ***cin.ignore(..)*** peuvent être ajoutées avant le retourner le choix de l'utilisateur pour vider complètement le tampon d'entrée pour les saisies à venir dans le programme.

Pour effacer l'écran sous Linux avant l'affichage du menu et en quittant la fonction un appel système peut être réalisé avec :

- sous Linux : system("clear");
- sous Windows : system("cls");

e) Pour la méthode **void Menu::AttendreAppuiTouche()** on propose le code suivant :

Méthode statique AttendreAppuiTouche() ***menu.cpp***

```
void Menu::AttendreAppuiTouche()
{
    string uneChaine;
    cout << endl << "appuyer sur la touche Entrée pour continuer...";
    getline(cin,uneChaine);
    cin.ignore( std::numeric_limits<streamsize>::max(), '\n' );
    system("clear");
}
```

f) Complétez le programme principal à partir des éléments suivants :

Programme principal ***main.cpp***

```
int main()
{
    int choix;
    Menu leMenu("menu.txt");
    do
    {
        choix = leMenu.Afficher();
        switch (choix)
        {
        case OPTION_1:
            cout << "Vous avez choisi l'option n°1" << endl;
            Menu::AttendreAppuiTouche();
            break;
        // à compléter
        }
    } while(choix != QUITTER);

    return 0;
}
```

```
enum CHOIX_MENU
{
    OPTION_1 = 1,
    OPTION_2,
    OPTION_3,
    OPTION_4,
    QUITTER
};
```

Remarque : Le fichier menu.txt doit être dans le même répertoire que l'exécutable de votre programme.

2.10. Traitement des erreurs avec une Classe d'exception

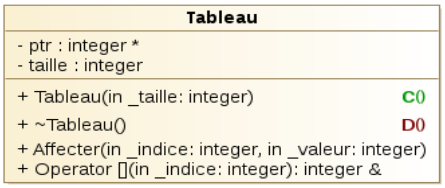
Afin d'améliorer et de simplifier le traitement des erreurs, il est possible de créer des classes susceptibles de gérer les exceptions. Prenons l'exemple d'une classe **Tableau** dont le rôle est de créer un tableau dynamiquement. Les risques de plantage peuvent être liés :

- à la création du tableau avec une dimension négative,
- à l'accès à un élément du tableau avec un indice négatif,
- et à l'accès en lecture ou en écriture à un élément du tableau d'indice supérieur ou égale à la taille du tableau, même si cela ne fait pas directement planter le programme, le risque est juste d'écraser des variables adjacentes dans la mémoire.

Un programme bien fait s'interdit toute ces opérations. Toutefois, rien n'empêche un programmeur de mal utiliser cette classe **Tableau**. Dans ce cas, il faut lui indiquer et, en général, stopper l'exécution. Pour cela, il est nécessaire de lever une exception lorsque les dimensions ne sont pas convenables ou que l'indice est en dehors des limites.

Définition de la classe Tableau *tableau.h*

```
#ifndef TABLEAU_H
#define TABLEAU_H

class Tableau
{
private:
    int *ptr;
    int taille;
public:
    Tableau(int _taille);
    ~Tableau();
    void Affecter(int _indice, int _valeur);
    int &operator[](int _indice);
};

#endif // TABLEAU_H
```



Comme nous pouvons le constater, la définition de la classe reste tout à fait classique.

Lors de l'implémentation de cette classe, il faut veiller à ce que les erreurs ne puissent pas se produire. Pour cela deux nouvelles classes peuvent être définies la première, la classe **ErreurCreation**, va réagir lors de problèmes de création du tableau, la seconde la classe **ErreurIndice** sera appelée lors d'un défaut d'accès aux éléments du tableau.

La définition de ces deux classes est réalisée sur le même modèle et peut être déclarée dans le fichier tableau.h.

Définition des classes d'exception *tableau.h*

```
class ErreurCreation
{
private:
    int codeErreur;
    string message;
public:
    ErreurCreation(int _codeErreur,
                   string _message);
    int ObtenirCodeErreur() const;
    string ObtenirDescription() const;
};
```

```
class ErreurIndice
{
private:
    int codeErreur;
    string message;
public:
    ErreurIndice(int _codeErreur,
                 string _message);
    int ObtenirCodeErreur() const;
    string ObtenirDescription() const;
};
```

Pour chaque type d'erreur, un code spécifique est déterminé ainsi qu'un message indiquant la nature de l'erreur. Les codes d'erreur peuvent être définis par des constantes énumérées.

Définition des constantes énumérée *tableau.h*

```
enum erreurs
{
    CREATION,
    INDICE
};
```

Ici les valeurs sont 0 et 1, mais rien n'empêche de fixer d'autres valeurs, 100 et 101 par exemple en fonction de la gestion d'erreurs pour le reste du programme.

L'implémentation de la classe Tableau devient alors :

Implémentation de la classe tableau *tableau.cpp*

```
Tableau::Tableau(int _taille):
    taille(_taille)
{
    if(taille <= 0)
    {
        ptr = nullptr;
        ErreurCreation excep(CREATION,"taille incorrecte lors de la création du tableau");
        throw (excep);
    }
    ptr = new int[taille];
}

Tableau::~Tableau()
{
    if(ptr != nullptr)
        delete[] ptr;
}
```

Dans le constructeur, si la taille de tableau est fixée de manière incorrecte, le pointeur est initialisé à ***nullptr***, qui représente la valeur du pointeur nul directement supportée par le langage C++11. Ensuite, une instance de la classe **ErreurCreation** est définie et l'exception est levée avec *throw*. L'absence du ***else*** ne fait pas défaut ici, car, dès l'instant où l'instruction ***throw*** est exécutée, le programme est dérouté vers le bloc ***catch*** correspondant à l'objet lancé. Reste simplement à l'utilisateur de cette classe à instancier la classe **Tableau** dans un bloc ***try*** qui sera présenté dans le programme principal. L'autre conséquence de l'exécution de l'instruction ***throw*** est l'appel de tous les destructeurs des objets déclarés dans le bloc ***try***. C'est pour cela qu'il est de bonne facture de tester si le pointeur **ptr** est bien différent de ***nullptr*** avant de libérer la mémoire, cette règle évite des plantages lorsque le programme s'arrête anormalement.

Remarque

L'exécution du programme reprend après le bloc ***catch*** et non pas à l'endroit où se trouve l'instruction ***throw*** en cas de levée d'exception.

La suite de l'implémentation de la classe Tableau reprend la même logique en utilisant la deuxième classe d'exception.

Implémentation de la classe Tableau *tableau.cpp*

```
void Tableau::Affecter(int _indice, int _valeur)
{
    if(_indice < 0 || _indice > taille)
    {
        ErreurIndice excep(INDICE,"L'indice du tableau n'est pas correct lors de l'affectation");
        throw (excep);
    }
    ptr[_indice] = _valeur;
}

int &Tableau::operator[](int _indice)
{
    if(_indice < 0 || _indice > taille)
    {
        ErreurIndice excep(INDICE,"L'indice du tableau n'est pas correct pour l'opérateur []");
        throw (excep);
    }
    return ptr[_indice];
}
```

Pour l'implémentation des deux classes d'exception, elles peuvent également être associées au code de la classe Tableau et réalisées dans le même fichier.

Implémentation des classes d'exception *tableau.cpp*

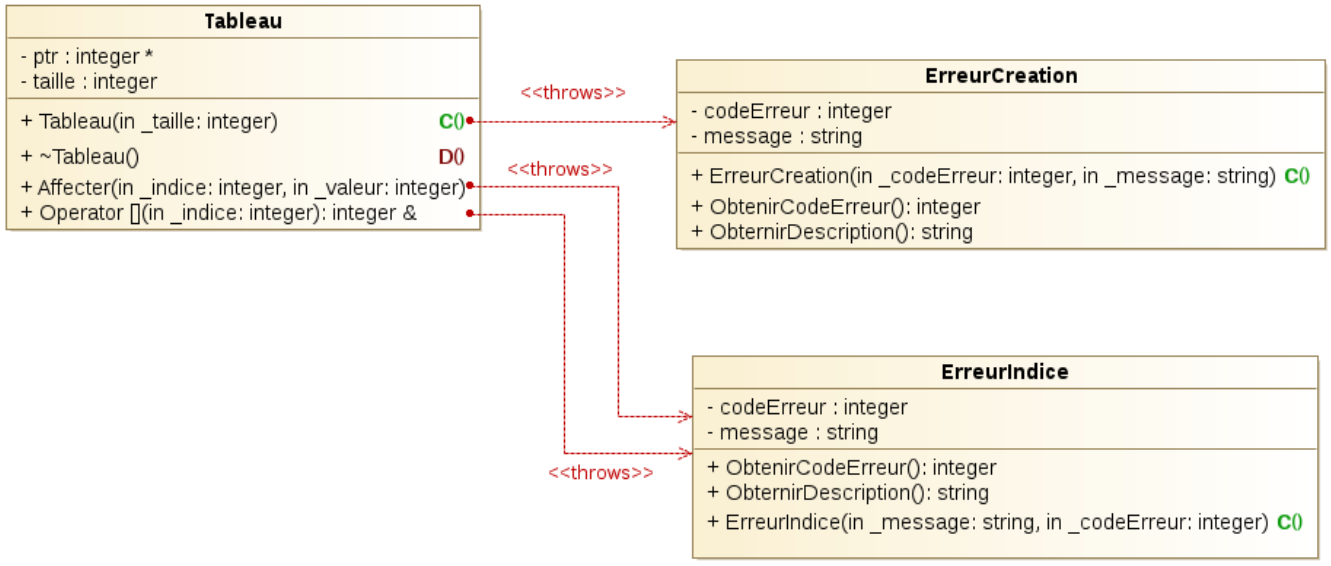
```
ErreurIndice::ErreurIndice(int _codeErreur, string _message):
    codeErreur(_codeErreur),
    message(_message)
{
}

int ErreurIndice::ObtenirCodeErreur() const
{
    return codeErreur;
}

string ErreurIndice::ObtenirDescription() const
{
    return message;
}

ErreurCreation::ErreurCreation(int _codeErreur, string _message):
    codeErreur(_codeErreur),
    message(_message)
{
}

int ErreurCreation::ObtenirCodeErreur() const
{
    return codeErreur;
}

string ErreurCreation::ObtenirDescription() const
{
    return message;
}
```

Elles sont construites, toutes deux, sur le même modèle.

Remarque

Par la suite, il pourra être envisagé d'améliorer ces classes d'exception en utilisant une classe abstraite, notion abordée dans le prochain chapitre.

La représentation UML de la classe **Tableau** et ces deux classes d'exception est donnée ici :



Pour finir cette étude sur les classes d'exception, il ne reste plus que l'implémentation du programme principal et notamment l'utilisation du bloc ***try*** et des blocs catch pour le traitement des exceptions.

Implémentation du programme principal ***utilisationtableau.cpp***

```
#include "tableau.h"
#include <iostream>
#include <stdlib.h>

using namespace std;

int main()
{
    try
    {
        Tableau leTableau(10);
        leTableau.Affecter(5,8);
        leTableau.Affecter(-5,15);
        cout << leTableau[-5] << endl;
        Tableau autre(-12);
    }

    catch(ErreurCreation const &exp)
    {
        cout << "Erreur " << exp.ObtenirCodeErreur() << endl;
        cout << exp.ObtenirDescription() << endl;
        exit (EXIT_FAILURE);
    }

    catch (ErreurIndice const &exp)
    {
        cout << "Erreur " << exp.ObtenirCodeErreur() << endl;
        cout << exp.ObtenirDescription() << endl;
        exit (EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Cette représentation montre bien les méthodes qui font appelles à une classe d'exception et laquelle.

Dans ce programme principal, les différents types d'erreurs ont été cumulés, lors de vos tests il est nécessaire de mettre en commentaire les lignes pour lesquelles on ne souhaite pas lever d'exception.

Remarque

Ici, le choix a été de stopper le programme, mais il aurait pu être envisagé tout autre cas de figure comme redemander une taille de tableau correcte...

Exercice d'application :

Reprenez la gestion de menu étudiée précédemment, recherchez quelles pourraient être les causes de dysfonctionnement de cette gestion et réalisez un traitement d'erreur approprié en utilisant une ou plusieurs classes d'exception.

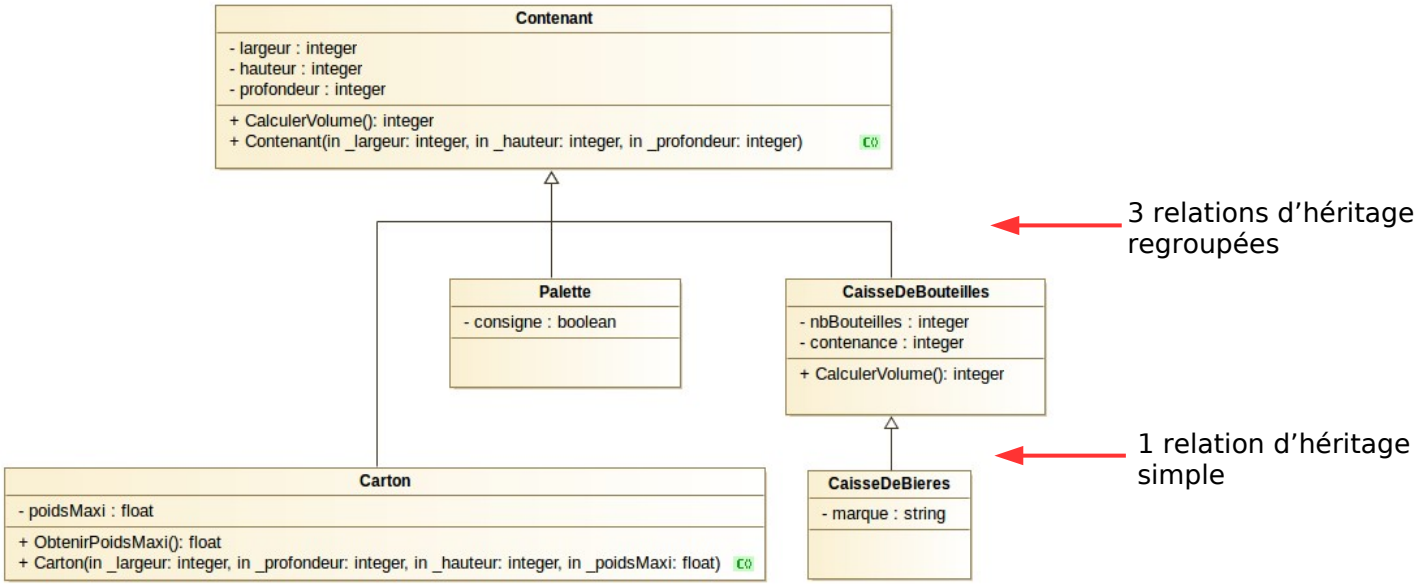
3. Relations entre objets

3.1. Mécanisme d'héritage

3.1.1. Introduction

Le concept d'héritage est l'un des fondements de la programmation orientée objet. Il permet de créer une nouvelle classe nommée **classe dérivée** à partir d'une classe existante, la **classe de base**. La classe dérivée hérite des caractéristiques de sa classe de base, données et fonctions membres dans une certaine mesure. Les seuls membres qui ne sont pas transmis sont les constructeurs, les destructeurs, les membres qui surdéfinissent l'opérateur d'affectation et les **fonctions amies**.

Cette relation traduit l'idée « **est une sorte de** ». Dans le modèle UML, elle est représentée par un triangle comme la montre la figure ci-après. Ce diagramme de classes est une vue incomplète des différentes classes de la hiérarchie.



L'héritage permet de **réutiliser** le code de la classe de base, tout en gardant la possibilité d'apporter les adaptations nécessaires à chaque cas particulier. Cette opération peut être réalisée avec ou sans les codes sources de la classe de base.

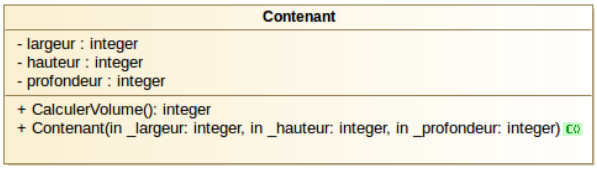
Lorsque le programmeur cherche à regrouper les éléments communs à plusieurs classes dans une même classe, on parle de **généralisation**. Le terme **spécialisation** est retenu lorsque les classes dérivent d'une classe de base.

Carton, **Palette** et **CaisseDeBouteilles** sont des sortes de **Contenant**. Ces classes spécialisent la classe **Contenant** en apportant chacun leurs spécificités. D'un autre point de vue, on peut dire que **Contenant** généralise les classes **Carton**, **Palette** et **CaisseDeBouteilles** en regroupant ce qui est commun aux trois classes.

La classe de base se traduit en C++ comme cela est décrit dans le chapitre précédent.

Déclaration de la classe Contenant *contenant.h*

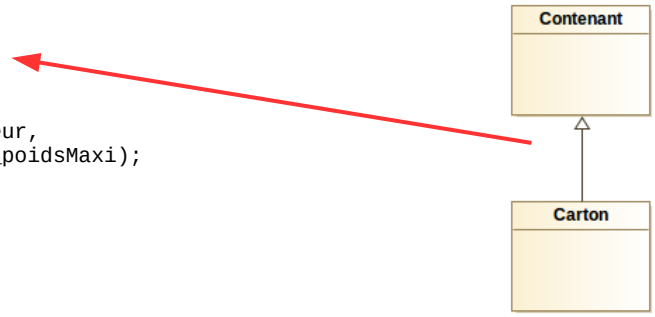
```
#ifndef CONTENANT_H
#define CONTENANT_H
class Contenant
{
public:
    Contenant(const int _largeur, const int _hauteur, const int _profondeur);
    int CalculerVolume();
private:
    int largeur;
    int hauteur;
    int profondeur;
};
#endif // CONTENANT_H
```



Pour la déclaration de la classe **Carton**, il faut maintenant tenir compte de la relation d'héritage.

Déclaration de la classe Carton *carton.h*

```
#ifndef CARTON_H
#define CARTON_H
#include "contenant.h"
class Carton : public Contenant
{
public:
    Carton(const int _largeur, const int _hauteur,
           const int _profondeur, const float _poidsMaxi);
    float ObtenirPoidsMaxi();
private:
    float poidsMaxi;
};
#endif // CARTON_H
```



La déclaration de la classe Carton est suivie de « ***: public*** **Contenant** » pour indiquer la relation d'héritage entre ces deux classes.

Le constructeur de la classe **Carton** reprend les paramètres de la classe **Contenant** car, il est chargé de les transmettre à celui de la classe de base.

Définition des opérations de la classe Contenant *contenant.cpp*

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

using namespace std;

Contenant::Contenant(const int _largeur, const int _hauteur, const int _profondeur):
    largeur(_largeur),
    hauteur(_hauteur),
    profondeur(_profondeur)
{
    cout << "constructeur de la classe Contenant" << endl ;
}


int Contenant::CalculerVolume()
{
    return largeur * hauteur * profondeur ;
}
```

Ici, les attributs sont initialisés en utilisant la liste d'initialisation. Cette méthode est à privilégier, elle est obligatoire pour les constantes et pour initialiser le constructeur d'une classe de base dans le cadre d'un héritage. Dans le cas présent, la définition du constructeur est équivalente au code présenté ci-dessous.

Autre façon de définir le constructeur de la classe Contenant *contenant.cpp*

```
Contenant::Contenant(const int _largeur, const int _hauteur, const int _profondeur)
{
    largeur = _largeur;
    hauteur = _hauteur;
    profondeur = _profondeur;
    cout << "constructeur de la classe Contenant" << endl ;
}
```

Exercice d'application

1. Après avoir ajouté un destructeur à ces deux classes, dont le rôle est simplement d'afficher de quel destructeur il s'agit, implémentez les deux classes **Contenant** et **Carton** en C++ (le code source de la classe **Carton** est donné à la page suivante).
2. Dans le programme principal, instanciez la classe **Carton** et relevez l'ordre de l'appel des constructeurs et des destructeurs.
3. À partir de la hiérarchie de classe présentée en introduction, réalisez la classe **CaisseDeBouteilles**, ajoutez un constructeur et un destructeur. Attention, la méthode **CaisseDeBouteilles ::CalculerVolume** retourne le volume contenu dans les bouteilles et non pas le volume brut du contenant.
4. Après avoir instancié un objet de type **CaisseDeBouteilles**, comment faire appel à la méthode **Contenant::CalculerVolume()** qui retourne le volume brut ? Testez l'appel dans le programme principal.

L'extrait de code suivant montre le passage de paramètre au constructeur de la classe **Contenant**. C'est toujours la première opération à effectuer lors de la définition du constructeur de la classe dérivée. Il reste ensuite à initialiser le ou les attributs de la classe dérivée.

Définition des opérations de la classe Carton *carton.cpp*

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

using namespace std;

Carton::Carton(const int _largeur, const int _hauteur, const int _profondeur, const float _poidsMaxi):
    Contenant(_largeur,_hauteur,_profondeur),
    poidsMaxi(_poidsMaxi)
{
    cout << "Constructeur de la classe Carton" << endl;
}

float Carton::ObtenirPoidsMaxi()
{
    return poidsMaxi;
}
```

Remarque

Lors d'un héritage, le constructeur de la classe de base est appelé en premier lieu, ce qui permet au constructeur de la classe dérivée de compléter les initialisations de l'objet. Pour les destructeur, ils sont appelés dans l'ordre inverse, le destructeur de la classe dérivée d'abord puis celui de la classe de base pour finir.

3.1.2. Modalités d'accès aux membres de la classe de base

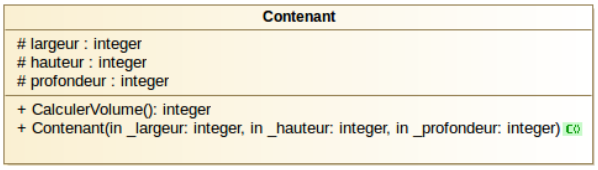
Dans l'exemple précédent, la classe dérivée n'a pas eu besoin d'accéder aux attributs de la classe de base. Si telle avait été le cas, cela n'aurait pas été possible. Le compilateur refuse systématiquement l'accès aux membres privés d'une classe même dans le cadre d'un héritage, c'est le principe de l'encapsulation.

Outre les statuts **public** et **privé** vus jusqu'à maintenant, il existe un statut **protégé**. Un membre protégé se comporte comme un membre privé pour une utilisation quelconque de la classe ou de la classe dérivée, mais comme un membre public pour la classe dérivée.

Pour la classe **Contenant**, on peut imaginer qu'il s'agit d'une erreur de conception. Dans le cadre d'une classe sujette à devenir une classe de base, le statut **protégé** sauf cas particulier est à privilégier. La Déclaration de la classe devient donc :

Déclaration de la classe Contenant *contenant2.cpp*

```
#ifndef CONTENANT_H
#define CONTENANT_H

class Contenant
{
public:
    Contenant(const int _largeur, const int _hauteur, const int _profondeur);
    int CalculerVolume();
protected:
    int largeur;
    int hauteur;
    int profondeur;
};

#endif // CONTENANT_H
```



On remarque dans la représentation UML de la classe le signe **#** devant chaque attribut, traduit en C++ par le mot clé **protected**.

L'utilisation du qualificateur **protected** ne fait pas bénéficier d'une encapsulation aussi complète que le qualificateur **private**, mais rend les attributs **largeur**, **hauteur** et **profondeur** accessibles dans les classes dérivées.

3.1.3. Dérivations publique, protégée et privée

Il est possible de contrôler les accès aux membres de la classe dérivée, quels que soient les choix qui avaient été effectués au niveau de la classe de base. En particulier, si celle-ci autorise des accès jugés trop libéraux.

Dérivation publique

Les membres de la classe de base conservent leur statut dans la classe dérivée. C'est la situation la plus usuelle.

Dérivation publique

```
class Carton : public Contenant ;
```

Dérivation protégée

Les membres publics de la classe de base deviennent membres protégés de la classe dérivée. Les autres membres conservent leur statut.

Dérivation publique

```
class Carton : protected Contenant ;
```

Dérivation privée

Tous les membres de la classe de base deviennent privés dans la classe dérivée. Par défaut, la dérivation est privée.

Dérivation publique

```
class Carton : private Contenant ;
```

ou

```
class Carton : Contenant ;
```

Par contre

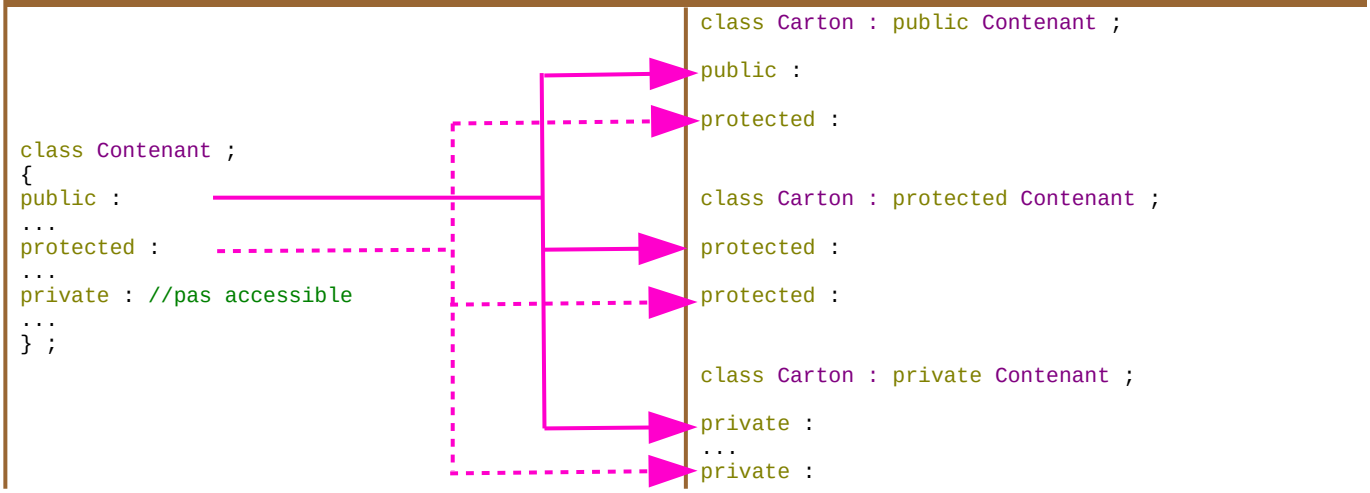
L'utilisation de dérivations protégées ou privées rend plus complexe la détermination du droit d'accès aux membres de chaque classe et donne donc un programme moins lisible.

De plus, les conversions implicites définies précédemment ne s'appliquent que pour l'héritage public.

Il est donc conseillé d'utiliser les dérivations « privée » et « protégée » avec prudence.

Voici un résumé du devenir des éléments de la classe de base dans la classe dérivée en fonction du type d'héritage

Résumé :



Exercice d'application

L'entreprise **mLego**, une fonderie à **Boëssé le sec** (72), est fabricant d'alliages de cuivre depuis 1894. Leur principale production est basée sur des barres de formes et de dimensions diverses en alliage de cuivre, d'aluminium et autres matières.

Chaque barre est caractérisée par une référence, une longueur, la densité et le nom de l'alliage qui la compose.

Étude de la classe *Barre* :

1. Déclarer et définir la classe **Barre** pertinente. Les paramètres pour l'initialisation seront passés au constructeur de la classe. La référence et le nom seront définis en utilisant le type ***string*** de la librairie standard. L'ensemble des attributs de cette classe sera privé.
2. Déclarer et définir une méthode **AfficherCaractéristiques** » en utilisant le flux de sortie standard de la librairie ***iostream***.

Études des différentes formes :

Comme le montre l'extrait du catalogue de la société, les barres sont de différentes formes rondes, hexagonales, rectangulaires, carrées, creuses, pleines...

3. Déterminer pour chacune de ces formes le nombre de paramètres nécessaire au calcul de la section. Répondre sous la forme d'un tableau.
4. Déclarer et définir les classes **BarreRonde**, **BarreRectangle** et **BarreCarree** disposant des attributs nécessaires pour calculer leur section respective. Chaque classe hérite de la classe de base **Barre**. Les paramètres d'initialisations des attributs seront passés par le constructeur et transmis le cas échéant au constructeur de la classe de base. Indiquer l'ordre de l'appel des constructeurs et des destructeurs.

Pour rappel, on donne ici les formules de calcul nécessaire pour les différentes formes de barre.

 Cercle	$S_{Cercle} = \Pi * d^2 / 4$	d représente ici, le diamètre du cercle
 Rectangle	$S_{Rectangle} = L * l$	L représente ici la longueur et l la largeur du rectangle. Pour un carré, la longueur et la largeur sont identiques.
 Hexagone	$S_{Hexagone} = 2\sqrt{(3*d_i^2/4)}$	d_i représente, ici, le diamètre du cercle inscrit de l'hexagone
 Octogone	$S_{Octogone} = 2 * d_i^2 *(\sqrt{2} - 1)$	d_i représente, ici, le diamètre du cercle inscrit de l'octogone

5. Ajouter à chaque sous-classe la méthode **CalculerSection** et son code.
6. Ajouter à chaque sous-classe une méthode **CalculerMasse** et son code. Pour rappel, la masse d'une barre est donnée par la formule :

   *masse = longueur * section * densité*

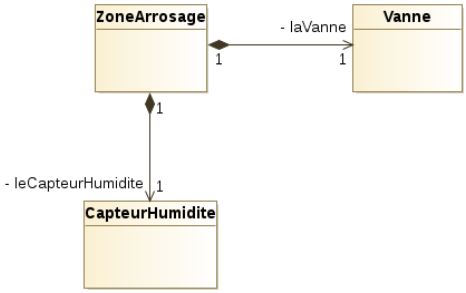
   Les attributs nécessaires à ce calcul dans la classe de base sont-ils accessibles dans la classe dérivée ? Réaliser la modification si nécessaire. Réaliser le codage de la méthode pour chaque classe déclarée précédemment.
7. Déclarer une instance de chaque type de barre et compléter le code du programme principal pour faire afficher les caractéristiques et la masse de chaque barre.

3.2. Mécanisme de composition

3.2.1. Introduction

La relation de composition implique une notion d'appartenance. Le cycle de vie des éléments composants est lié à celui de l'agrégat, ou élément contenant. Si l'agrégat est détruit ou copié, ses composants le sont aussi. À un même instant une instance composante ne peut être liée qu'à un seul et unique agrégat.

Cette relation traduit l'idée « **est constituée physiquement de** ». Dans le modèle UML, elle est représentée par un losange noir plein, comme le montre la figure ci-après. La relation est généralement orientée, le sens de la flèche indique la navigabilité, de la classe agrégat, celle du côté losange, vers la classe agrégée.



L'exemple ci-contre représente une étude partielle d'un contrôleur de serre chargé de l'arrosage. Ici, une zone d'arrosage est composée d'une vanne et d'un capteur d'humidité.

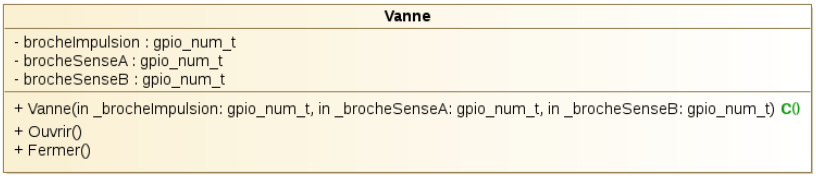
La relation de composition est représentée en UML par le losange plein. Les deux relations étant orientées, c'est bien la classe **ZoneArrosage** qui contient un objet nommé **laVanne** une instance de la classe **Vanne** et un autre objet nommé **leCapteurHumidite** une instance de la classe **CapteurHumidite**. Le nombre d'instances est déterminé par la cardinalité, ici 1 pour chacune des relations.

Les rôles, noms donnés à **laVanne** et **leCapteurHumidite**, représentent deux attributs de la classe **ZoneArrosage,** ils ne sont pas directement représentés dans la classe agrégat, comme le montre la figure.

3.2.2. Implémentation en C++

Comme la durée de vie d'un composant dépend de la classe agrégat, c'est cette dernière qui doit créer l'instance du composant et la détruire. Elle peut être déclarée de manière automatique ou de manière dynamique.

Le détail de la classe **Vanne** est donné sous sa représentation **UML** ci-contre. Le type **gpio_num_t** désigne une broche d'entrée / sortie sur un microcontrôleur type **ESP32**, il est équivalent à un entier désignant un numéro de broche.



La déclaration de la classe **Vanne** et de la classe **CapteurHumidite** est tout à fait classique, voici celle de la classe **Vanne**.

Déclaration de la classe Vanne ***vanne.h***

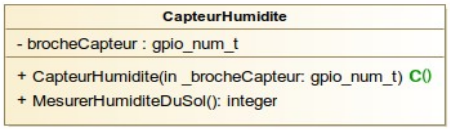
```
#ifndef VANNE_H
#define VANNE_H
class Vanne
{
public:
    Vanne(const gpio_num_t _brocheImpulsion,const gpio_num_t _sensA,const gpio_num_t _sensB);
    void Ouvrir();
    void Fermer();
private:
    gpio_num_t impulsion;
    gpio_num_t sensA;
    gpio_num_t sensB;
};
#endif // VANNE_H
```

Pour la compilation dans un environnement C++ classique, gpio_num_t peut être redéfini en int

#define gpio_num_t int

Voici la représentation **UML** de la classe **CapteurHumidite**. La méthode **MesurerHumiditeDuSol** retourne un entier, le pourcentage d'humidité mesuré dans le sol de la serre.

Exercice d'application



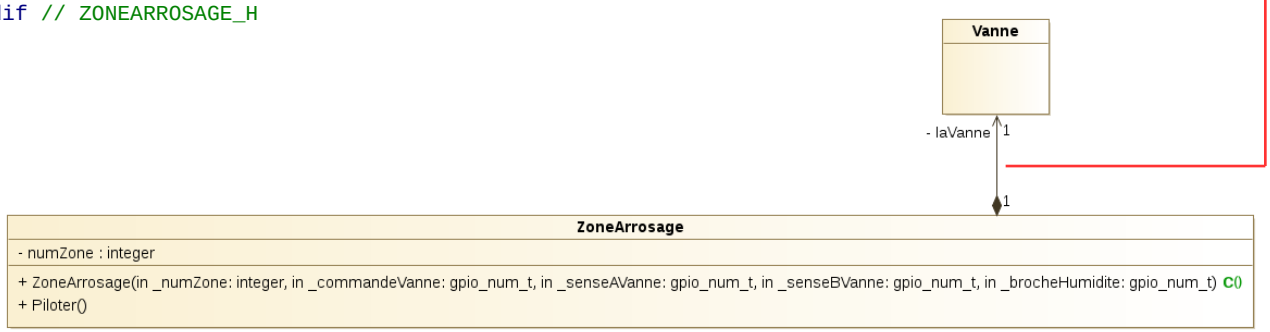
1. À titre d'exercice, réalisez la déclaration de cette classe en C++.

Implémentation de la relation sous la forme d'une instance automatique :

Dans ce cas de figure, l'attribut représentant le composant est une simple instance. Pour l'exemple, seule la composition avec la classe **Vanne** est prise en considération.

Déclaration de la classe ZoneArrosage — *zonearrosage.h*

```
#ifndef ZONEARROSAGE_H
#define ZONEARROSAGE_H

#include "vanne.h"

class ZoneArrosage
{
private:
    Vanne laVanne;
    int numZone;
public:
    ZoneArrosage(int _numZone,
                 const gpio_num_t _commandeVanne,
                 const gpio_num_t _senseAVanne,
                 const gpio_num_t _senseBVanne,
                 const gpio_num_t _brocheHumidite);
    void Piloter();
};

#endif // ZONEARROSAGE_H
```



Dans le cas présent, l'instanciation n'est pas suffisante, car le constructeur de la classe Vanne possède trois paramètres qu'il faut initialiser. Le passage de paramètres ne pouvant se faire dans le fichier d'en-tête *zonearrosage.h*, il doit être fait lors de l'implémentation du constructeur.

Implémentation du constructeur de la classe ZonneArrosage ***zonearrosage.cpp***

```
ZoneArrosage::ZoneArrosage(const int _numZone, const int _commandeVanne, const int _senseAVanne,
                           const  int _senseBVanne, const int _brocheHumidite):
    laVanne(_commandeVanne,_senseAVanne,_senseBVanne),
    numZone(_numZone)
{
    // reste du code pour le constructeur
}
```

Le passage de paramètres doit se faire **impérativement** avec la liste d'initialisation comme cela avait été le cas dans la relation d'héritage.

2. Complétez la déclaration de la classe **ZoneArrosage** pour faire apparaître la deuxième relation de composition avec la classe **CapteurHumidite**.
3. Modifiez l'implémentation du constructeur afin de prendre en compte cette deuxième relation.

À retenir

Lorsque le constructeur du composant ne possède pas de paramètre, il peut être instancié simplement dans la classe agrégat. Dans le cas contraire, comme l'a montré l'exemple, il est nécessaire de passer les paramètres au travers de la liste d'initialisation.

Implémentation de la relation sous la forme d'une instance dynamique :

Cette fois-ci, l'attribut ne représente pas directement une instance du composant. La relation va être implémentée sous la forme d'un pointeur que le constructeur **doit initialiser dynamiquement.** Un destructeur est également nécessaire pour libérer la mémoire allouée dans le constructeur. En procédant de la sorte, le comportement reste identique, le constructeur crée le composant et le destructeur le détruit, à la différence que c'est, ici, le programmeur qui prend à sa charge ces opérations.

Dans la déclaration ci-après, **laVanne** est maintenant un pointeur sur la classe **Vanne**, pointeur qu'il est nécessaire d'initialiser dans le constructeur.

Certains peuvent se demander, pourquoi faire aussi compliqué puisque la première version fonctionnait très bien sans intervention du programmeur ?

Parfois, le constructeur de l'agrégat ne possède pas directement les éléments pour le passage de paramètres au constructeur du composant. Ces éléments peuvent être lus dans un fichier ou obtenus sous une autre forme, dans ce cas l'implémentation sous la forme d'un pointeur s'avère très utile.

Déclaration de la classe ZoneArrosage ***zonearrosage.h***

```
#ifndef ZONEARROSAGE_H
#define ZONEARROSAGE_H

#include "vanne.h"

class ZoneArrosage
{
private:
    Vanne *laVanne;
    int numZone;
public:
    ZoneArrosage(const string _initialisationZone);
    void Piloter();
};

#endif // ZONEARROSAGE_H
```

Ici, les paramètres sont passés sous la forme d'une chaîne de caractères au lieu d'être transmis sous la forme de plusieurs entiers comme dans la version précédente.

Le programme principal peut alors s'écrire de la manière suivante :

Programme principal *serre.cpp*

```
#include "zonearrosage.h"
int main(int argc, char *argv[])
{
    ZoneArrosage zone1("1 25 15 5"); // zone 1, impulsion sur 25, sens A=15 et B=5
}
```

Et, voici le code du constructeur : il est chargé de décomposer la chaîne et instancié la classe **Vanne**.

Implémentation du constructeur *zonearrosage.cpp*

```
ZoneArrosage::ZoneArrosage(const string _initialisationZone)
{
    int parametres[4];
    size_t prec = 0;
    size_t pos = 0;
    for (int indice = 0;indice < 4;indice++)
    {
        pos = _initialisationZone.find(' ',prec);
        parametres[indice] = atoi(_initialisationZone.substr(prec,pos).c_str());
        prec = pos + 1; // on incrémente pos pour le tour d'après.
    }
    numZone = parametres[0];
    laVanne = new Vanne(parametres[1],parametres[2],parametres[3]);
}
```

Le destructeur libère simplement la mémoire allouée dynamiquement.

Destructeur de la classe ZoneArrosage *zonearrosage.cpp*

```
ZoneArrosage::~ZoneArrosage()
{
    delete laVanne;
}
```

À retenir

Dans cette deuxième version pour l'implémentation de la relation de composition, le programmeur prend la charge de créer l'instance du composant dynamiquement et de restituer la mémoire au sein de la classe agrégat.

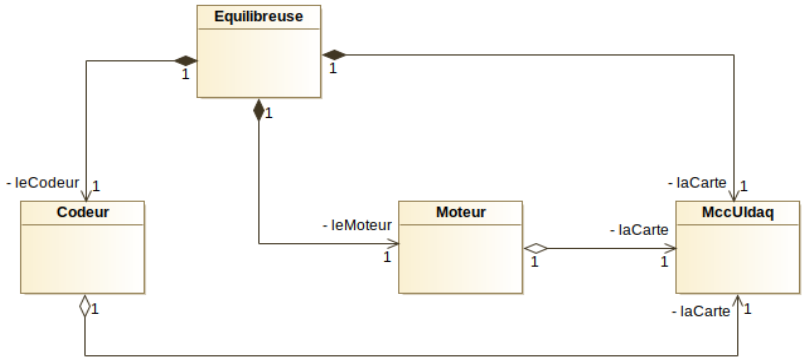
4. Ajoutez les éléments nécessaires pour implémenter la deuxième relation de composition avec la classe **CapteurHumidite** de manière dynamique. Dans le programme principal, ajouter un nouveau paramètre à la chaîne d'initialisation par exemple 12 qui représente la broche utilisée par le capteur d'humidité.
5. Dans chaque constructeur, affichez vers la sortie standard le nom de la classe et la valeur des attributs afin de vérifier le bon fonctionnement de votre implémentation.

3.3. Mécanisme d'agrégation.

3.3.1. Introduction

L'agrégation représente une relation de subordination et exprime un couplage fort. Il est cependant moins important que dans la relation de composition. Dans ce cas de figure, la durée de vie des objets « agrégat » et « agrégé » est indépendante. L'instance d'une classe agrégée peut être liée à d'autres objets à un même moment. L'agrégation se représente en UML par le losange vide. Pour la relation de composition, on pouvait parler d'agrégation forte ou par valeur.

L'exemple suivant présente un extrait de l'étude d'une maquette d'équilibrage. La classe **Equilibreuse** est composée d'une instance **leMoteur** d'une classe **Moteur** et d'une instance **leCodeur** d'une classe **Codeur**. Dans le même temps, chacune des classes **Moteur** et **Codeur** utilise une classe nommée **MccUldaq**. Cette dernière implémente des méthodes pour piloter physiquement le moteur et lire les informations en provenance du codeur, mais elle ne doit être instanciée qu'une seule fois. C'est donc la classe **Equilibreuse** qui va réaliser cette implémentation, elle est physiquement composée de cette carte d'acquisition, et les deux autres classes posséderont une relation d'agrégation avec cette classe. La figure suivante représente une vue partielle du diagramme de classes UML.



On remarque, les 3 relations de composition, les losanges noirs pleins et les 2 relations d'agrégation, les losanges vides.

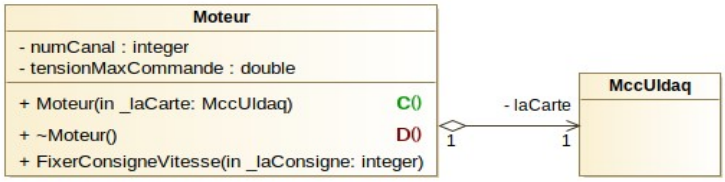
Dans les cas, les relations sont orientées, de la classe agrégat vers une classe agrégée. La cardinalité est également à 1, ce qui signifie : une seule instance.

3.3.2. Implémentation en C++

La relation d'agrégation s'implémente sous la forme d'une référence en C++.

Ainsi, la classe **Moteur** et la classe **Codeur** disposeront chacune d'une référence sur la classe **MccUldaq**. Cela reprend bien l'idée d'un alias vers cette classe. La classe **Equilibreuse** crée l'instance **MccUldaq** en premier lieu et en transmet une référence au constructeur de ses deux autres composants. La classe **Moteur** et la classe **Codeur** doivent posséder chacune un attribut nommé **laCarte**, comme l'indique le rôle sur la relation d'agrégation, de type référence sur la classe **MccUldaq.**

Le détail de la classe **Moteur** est représenté en UML par la figure suivante :



De la même manière que pour la composition, l'attribut **laCarte** n'apparaît pas avec les attributs **numCanal** et **tensionMaxCommande** puisqu'il est déjà représenté sous la forme d'un rôle au niveau de la relation.

La déclaration en C++ de cette relation peut se traduire de la manière suivante :

Déclaration de la classe Moteur *moteur.h*

```
#ifndef MOTEUR_H
#define MOTEUR_H

class MccUldaq;

class Moteur
{
private:
    int numCanal ;
    double tensionMaxCommande ;
    const MccUldaq & laCarte; // Pour la relation d'agrégation,
                              // l'attribut est ici constant pour être en accord avec le passage de
                              // paramètres. Il n'est pas modifié dans la classe Moteur.
public:
    Moteur(const MccUldaq & _laCarte, const int _numCanal, const double _tensionMaxCommande);
    ~Moteur();
    void FixerConsigne(const int _laConsigne);
};

#endif // MOTEUR_H
```

À noter, pour éviter tous problèmes d'inclusion circulaire, le fichier *MccUldaq.h* n'a pas été inclus dans le fichier *moteur.h*. On y trouve simplement le fait que **MccUldaq** est une classe.

Pour ce qui concerne la classe **Equilibreuse**, il s'agit de compositions, on propose ici d'implémenter de manière automatique la classe **MccUldaq** et de manière dynamique les deux autres. On s'intéresse, dans un premier temps, aux constructeurs des classes différentes classes.

Déclaration de la classe Equilibreuse *equilibreuse.h*

```
#ifndef EQUILIBREUSE_H
#define EQUILIBREUSE_H


#include "moteur.h"
#include "MccUldaq.h"


class Equilibreuse
{
private:
    MccUldaq laCarte;
    Moteur *leMoteur;
public:
    Equilibreuse();
};


#endif // EQUILIBREUSE_H
```

Le constructeur de la classe **Equilibreuse** se traduira simplement de la manière suivante :

Constructeur de la classe Equilibreuse *equilibreuse.cpp*

```
#include "equilibreuse.h"
Equilibreuse::Equilibreuse()
{
    leMoteur = new Moteur(laCarte);
}
```

Il ne reste plus qu'a étudier le constructeur de la classe **Moteur** pour voir l'ensemble du processus.

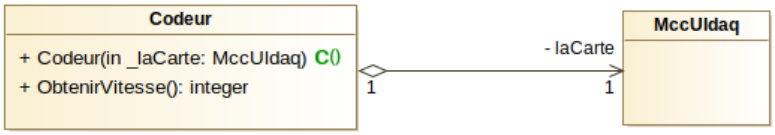
Constructeur de la classe Moteur *moteur.cpp*

```
#include "moteur.h"
#include "MccUldaq.h"

Moteur::Moteur(const MccUldaq &_laCarte, const int _numCanal, const double _tensionMaxCommande) :
    numCanal(_numCanal),
    tensionMaxCommande(_tensionMaxCommande),
    laCarte(_laCarte) // initialisation de la relation d'agrégation
{
    laCarte.ulAOut(numCanal,0);
}
```

Le fichier ***MccUldaq.h*** est inclus ici, pour que la classe **Moteur** puisse faire appel au différentes méthodes de la classe **MccUldaq** et palier au fait qu'il n'a pas été inclus dans le fichier ***moteur.h***.

Exercice d'application

1. Réalisez la déclaration de la classe **Codeur** dont la représentation UML est donnée ici.



2. Complétez la déclaration de la classe **Equibreuse** pour réaliser la relation avec la classe **Codeur**.
3. Réalisez le code du constructeur de la classe **Codeur** et complétez celui de la classe **Equilibreuse**.
4. Un destructeur explicite est-il nécessaire dans la classe **Equilibreuse** ? Si oui, complétez la déclaration de la classe **Equilibreuse** et codez ce destructeur.

À retenir

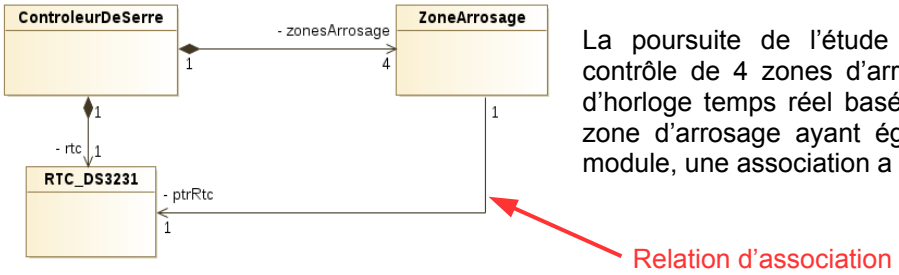
Dans le cas de la relation d'agrégation, l'objet agrégé n'est pas instancié dans le constructeur. Il est simplement initialisé à l'aide d'un paramètre passé au constructeur.

3.4. Mécanisme d'association

3.4.1. introduction

La relation d'association est similaire à la relation d'agrégation vue précédemment dans le sens ou la durée de vie des objets n'est pas liée. Elle indique cependant un lien moins fort que l'agrégation et par conséquent que la composition. La symbolique UML pour la représenter est un simple trait entre deux classes. Elle peut être navigable dans les deux sens ou uniquement dans un seul, dans ce cas une flèche marque le sens de la navigabilité.

Pour illustrer ce propos, l'exemple du contrôleur de serre va être approfondi :



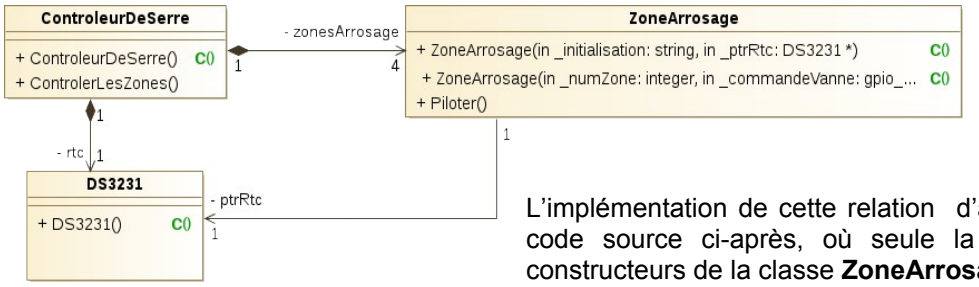
La poursuite de l'étude montre qu'un contrôleur assure le contrôle de 4 zones d'arrosage et qu'il dispose d'un dispositif d'horloge temps réel basé sur un composant DS3231. Chaque zone d'arrosage ayant également besoin de faire appel à ce module, une association a été mise en place vers ce composant.

3.4.2. Implémentation en C++

L'implémentation de la relation d'association est réalisée par un pointeur, comme elle peut l'être avec la relation de composition, mais l'instanciation est faite par ailleurs comme pour la relation d'agrégation.

La classe **ControleurDeSerre** va créer une instance nommée **rtc** de la classe **RTC_DS3231** puis 4 instances implémentées sous la forme d'un tableau nommé **zonesArrosage** de la classe **ZoneArrosage**. En effet, on constate que la cardinalité pour la relation vers la classe **ZoneArrosage** est égale à 4, d'où l'utilisation d'un tableau pour stocker les 4 instances. Le constructeur de la classe **ZoneArrosage** doit également recevoir un pointeur sur la classe **RTC_DS3231** pour initialiser son attribut **ptrRtc**.

La représentation UML des différentes classes est montrée à la figure ci-après :



L'implémentation de cette relation d'association est donnée dans le code source ci-après, où seule la deuxième implémentation du constructeurs de la classe **ZoneArrosage** été codée.

Déclaration de la classe ZoneArrosage *zonearrosage2.h*

```
#ifndef ZONEARROSAGE_H
#define ZONEARROSAGE_H

#include "vanne.h"
class DS3231;

class ZoneArrosage
{
private:
    const DS3231 *ptrRTC;
    int numZone;
public:
    ZoneArrosage(const string _initialisationZone,const DS3231 * _ptrRTC);
    void Piloter();
};

#endif // ZONEARROSAGE_H
```

La déclaration de la classe **ControleurDeSerre** montre l'implémentation des deux compositions, dont une multiple pour la classe **ZoneArrosage**, la cardinalité est de 4 dans le diagramme de classes.

Déclaration de la classe ControleurDeSerre *controleurdeserre.h*

```
#ifndef CONTROLEURDESERRE_H
#define CONTROLEURDESERRE_H

#include "zonearrosage.h"
#include "DS3231.h"

class ControleurDeSerre
{
private:
    DS3231 rtc;
    ZoneArrosage * zonesArrosage[4];
public:
    ControleurDeSerre();
    ~ControleurDeSerre();
    void ControlerLesZones();
};

#endif // CONTROLEURDESERRE_H
```

Un tableau de pointeur a été utilisé ici afin de réaliser une initialisation dynamique des 4 instances.

Le constructeur est chargé d'instancier 4 fois la classe **ZoneArrosage**. L'initialisation est réalisée à partir des données lues dans un fichier texte.

Implémentation du constructeur et du destructeur de ControleurDeSerre *controleurdeserre.cpp*

```
#include <fstream>
#include <iostream>
#include <string>
#include "controleurdeserre.h"

using namespace std;

ControleurDeSerre::ControleurDeSerre()
{
    ifstream fichier("config.txt");
    if (!fichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;
    else
    {
        string init;
        for (int indice = 0;indice < 4; indice++)
        {
            getline(fichier,init);
            if(fichier.good())
                zonesArrosage[indice] = new ZoneArrosage(init,&rtc);
        }
        fichier.close();
    }
}

ControleurDeSerre::~ControleurDeSerre()
{
    for (int indice = 0;indice < 4; indice++)
    {
        delete zonesArrosage[indice];
    }
}
```

Transmission de l'adresse de l'instance rtc pour finaliser l'association

Pour la classe **ZoneArrosage**, le constructeur reste identique à la version présentée précédemment en ajoutant l'initialisation du pointeur pour finaliser la relation d'association avec la classe **DS3231**.

Implémentation du constructeur de la classe ZoneArrosage *zonearrosage2.cpp*

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

using namespace std;

ZoneArrosage::ZoneArrosage(const string _initialisation,const DS3231 * _ptrRTC):
    ptrRTC(_ptrRTC)
{
    int parametres[4];
    size_t prec = 0;
    size_t pos =  0;
    for (int indice = 0;indice < 4;indice++)
    {
        pos = _initialisation.find(' ',prec);
        parametres[indice] = atoi(_initialisation.substr(prec,pos).c_str());
        prec = pos + 1; // on incrémente pos pour le tour d'après.
    }
    numZone = parametres[0];
    cout << "Zone " << numZone << "  ";
    laVanne = new Vanne(parametres[1],parametres[2],parametres[3]);
}
```

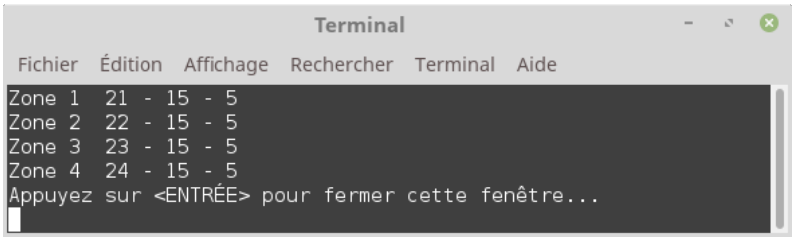
Initialisation du pointeur

Pour les besoins de la compilation, la classe DS3231 peut être définie de la manière suivante :

Compléments pour la compilation

```
----- Fichier ds3231.h ------

#ifndef DS3231_H
#define DS3231_H

class DS3231
{
public:
    DS3231();
};


#endif // DS3231_H
```

```
----- Fichier ds3231.cpp ------

#include "ds3231.h"
DS3231::DS3231()
{

}
----- Fichier main.cpp ------
#include "controleurdeserre.h"

int main(int argc, char *argv[])
{
    ControleurDeSerre controleur;
}
```



Remarque

L'utilisation du premier constructeur de la classe **ZoneArrosage** doit également être modifié, car jusqu'à présent il n'initialise pas le pointeur sur la classe DS3231. L'appel d'une de ces méthodes pourraient engendrer un plantage du programme. **Un pointeur, comme toute variable, doit toujours être initialisé avant son utilisation.**

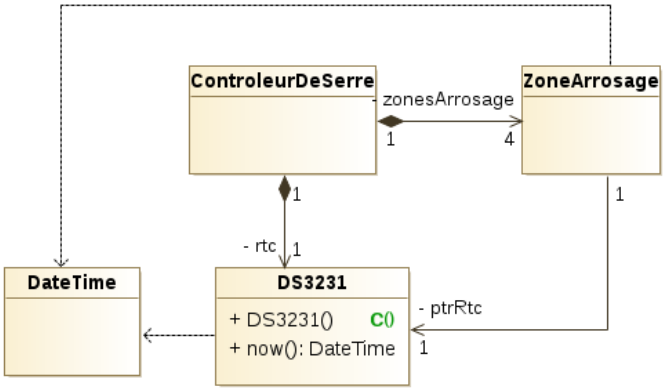
Exercice d'application

1. Modifiez et complétez le code de ce 1er constructeur vu lors de l'implémentation automatique de la relation de composition.

3.5. Notion de dépendance

Une dernière relation peut apparaître dans les diagrammes de classe, c'est la relation de dépendance. Elle exprime le fait que deux classes ne nécessitent pas forcément de lien structuré entre leurs objets. Lorsque cette relation est réalisée, elle est limitée dans le temps. Typiquement, cette relation existe lorsqu'une méthode déclare localement un nouvel objet et n'est donc pas attribut de la classe.

La relation est représentée par une flèche avec un trait en pointillé. La Classe **DS3231** utilise pour fournir la date et l'heure courante une instance de la classe **DateTime** dans la méthode **now()**. De même, chaque zone d'arrosage a besoin de connaître cette heure courante pour déclencher ou arrêter l'arrosage. Une de ces méthodes utilise donc également une instance de la classe **DateTime**.



Cette relation a tendance à surcharger les diagrammes et donc rendre moins lisibles. Chaque fois qu'une méthode utilise une instance d'une autre classe, il y a relation de dépendance. Elle doit apparaître uniquement pour montrer un point important que le concepteur souhaite souligner.

À retenir

La relation de dépendance n'est généralement pas représentée dans un diagramme de classes pour éviter de le surcharger. Elle se traduit en C++ par une instance locale à une méthode, en aucun cas par un attribut.

4. Fonction virtuelle, polymorphisme, classe abstraite

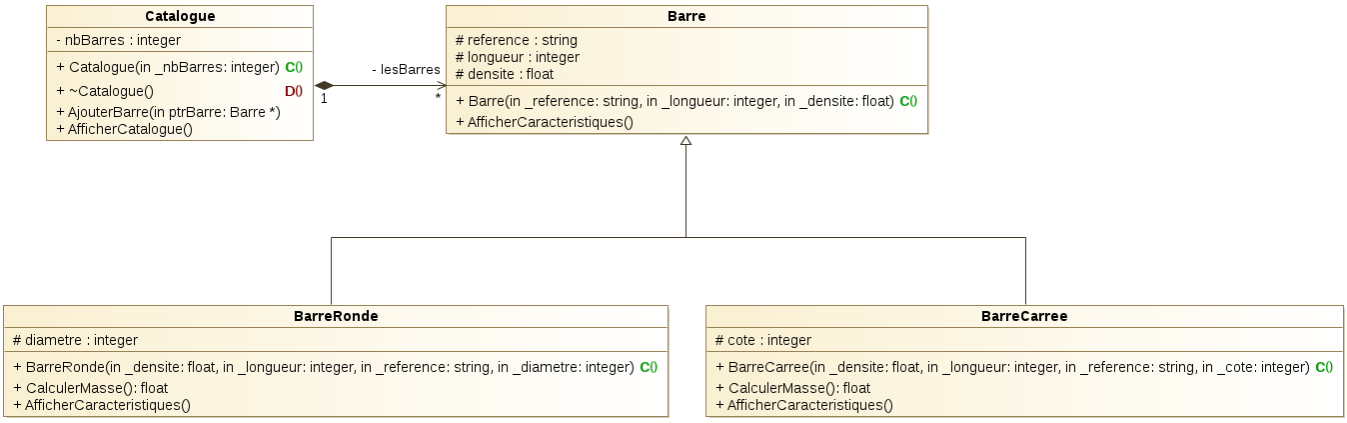
Note au lecteur

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*

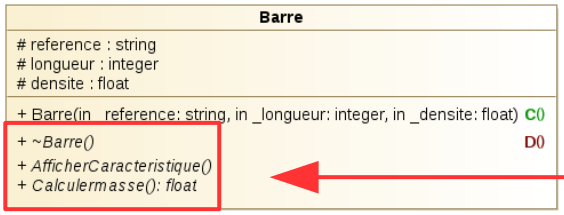
```
#ifndef BARRE_H
#define BARRE_H

#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

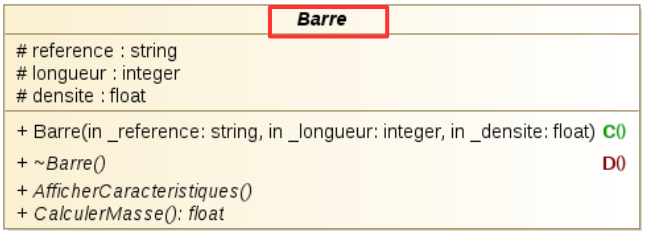
Une fonction virtuelle pure se déclare avec une initialisation à zéro. C'est ce qui aurait pu être fait avec la nouvelle fonction **CalculerMasse( )** de la classe **Barre** :

```
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

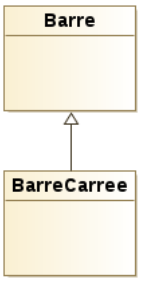


Une classe dérivée qui ne fournit pas une implémentation pour toutes les fonctions virtuelles pures de sa classe de base reste toujours une classe abstraite et ne peut donc pas être directement instanciée.

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

Choix n°3

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...
```

5. Programmation générique : les « templates »

5.1. Introduction

Jusqu'à présent, on a vu que les méthodes pouvaient être surchargées pour s'adapter à différents types de données, que les classes abstraites pouvaient servir de modèle pour leur classe dérivée. C'est ces différents concepts que nous allons approfondir maintenant avec l'idée de chercher à réduire l'écriture du code. Prenons l'exemple de la fonction suivante :

Fonction RechercherPlusPetit *pluspetit.cpp*

```
int RechercherPlusPetit(int a, int b)
{
    int retour;
    if (a < b)
        retour = a;
    else
        retour = b;
    return retour;
}
```

L'objectif est de retrouver le plus petit des deux nombres entiers, on peut bien sûr la surcharger pour obtenir la même chose avec des réels, des caractères…

L'idée est de développer une fonction générique qui s'adapte au type qu'on lui applique sans réécrire de code.

5.2. Fonctions modèles ou « template »

Dans un premier temps, il est nécessaire de définir un type de **variable générique**. Il peut représenter n'importe quel autre type. Puis dans un deuxième temps de réaliser la fonction en utilisant ce type générique. La syntaxe pour déclarer le type générique utilise les signes inférieur et supérieur et n'est pas terminée par un point virgule.

Fonction RechercherPlusPetit *fonctiontemplate.cpp*

```
template <typename T> // déclaration du type générique T
T RechercherPlusPetit(const T& a, const T& b)
{
    T retour;
    if (a < b)
        retour = a;
    else
        retour = b;
    return retour;
}
```

Le compilateur va générer automatiquement toutes les fonctions dont vous avez besoin à partir de leur utilisation. Lors de l'appel de la fonction, il faut juste préciser le type réel avec lequel elle doit travailler.

Programme principal *fonctiontemplate.cpp*

```
int main()
{
    double densiteAl(2.7);
    double densiteCu(8.9);

    cout << RechercherPlusPetit<double>(densiteCu,densiteAl) << endl;

    int parking(-1);
    int terrasse(5);

    cout << RechercherPlusPetit<int>(parking,terrasse) << endl;

    return 0;
}
```

Le type à traiter est également mis entre les signes inférieur et supérieur après le nom de la fonction.

On peut également imaginer une fonction qui calcule la moyenne des éléments d'un tableau. Cette fonction possède deux paramètres, elle peut s'écrire de la manière suivante :

Fonction CalculerMoyenne *fonctiontemplate.cpp*

```
template <typename T>
T CalculerMoyenne(T tab[], int nbElements)
{
    T somme = 0;
    for (int indice = 0; indice < nbElements; indice++)
    {
        somme += tab[indice];
    }
    return somme / nbElements;
}
```

La fonction peut être appelée de la manière suivante :

programme principal *fonctiontemplate.cpp*

```
int main()
{
    double temperature[5] = {12.5,25.2,14.4,18.3,20.1};
    int note[5]={11,10,13,15,14};

    cout << CalculerMoyenne<double>(temperature,5) << endl;
    cout << CalculerMoyenne<int>(note,5) << endl;

    return 0;
}
```

On remarque que tout semble normal dans le premier cas, mais pour le second une question se pose : le résultat attendu doit-il être sous la forme d'un entier ou d'un réel ?

Cette première version a retourné un nombre du même type que celui fourni à la fonction.

Avec les fonctions modèles, il est tout à fait envisageable de préciser plusieurs types génériques et ainsi avoir un paramètre de retour différent par exemple :

Fonction CalculerMoyenne *fonctiontemplate.cpp*

```
template <typename T, typename R>
R CalculerMoyenneV2(T tab[], int nbElements)
{
    R somme=0;
    for (int indice = 0; indice<nbElements; indice++)
    {
        somme += tab[indice];
    }
    return somme/static_cast<int>(nbElements);
}
```

La ligne : **cout << CalculerMoyenneV2<int,double>(note,5) << endl;** dans le programme principal donne comme résultat cette fois-ci 12,6 au lieu de 12.

À retenir

Les types génériques peuvent s'appliquer à n'importe quel type ou objet dès l'instant ou l'opérateur utilisé dans la fonction a été surchargé. Par exemple l'opérateur **<** pour la fonction **RecherPlusPetit()** ou l'opérateur **+** dans la fonction **CalculerMoyenne()**.

5.3. Les classes templates

Les classes templates, dans le même ordre d'idée, sont des classes dont le type des arguments peut varier. Dans le module ***Structure et gestion de données***, la structure de type Pile a pu être étudiée. Cette pile était typée à un seul objet, des entiers par exemple. Ce paragraphe va étendre la notion de Pile à toutes sortes d'objets sans pour autant multiplier le code.

Comme pour les fonctions, il faut définir le type générique, puis définir la classe classiquement en utilisant le type générique là où cela est nécessaire. L'exemple ci-dessous définit une pile de manière minimaliste.

Classe Pile *pile.h*

```
#ifndef PILE_H
#define PILE_H

template <typename T>
class Pile
{
private:
    int taille ;
    int sommet ;
    T *laPile;
public:
    Pile(const int _taille=10);
    void Empiler(const T element);
    T Depiler();
    bool PileVide();
};
```

Jusqu'ici rien de particulier, pour la suite, les méthodes doivent également être impérativement définies dans le fichier d'en-tête **pile.h** sinon le compilateur refuse de faire son travail.

Implémentation des méthodes *pile.h*

```
template<typename T>
Pile<T>::Pile(const int _taille):
    taille(_taille),
    sommet(0)
{
    laPile = new T[taille];
}

template<typename T>
T Pile<T>::Depiler()
{
    T retour;
    if(!PileVide())
        retour = laPile[--sommet];
    return retour;
}

template<typename T>
Pile<T>::~Pile()
{
    delete[] laPile;

}
```

```
template<typename T>
bool Pile<T>::PileVide()
{
    bool retour = false;
    if(sommet == 0)
        retour = true;
    return retour;
}

template<typename T>
void Pile<T>::Empiler(const T element)
{
    if(sommet < taille)
        laPile[sommet++] = element;
}
#endif // PILE_H
```

Le type générique doit être repris devant chaque méthode. Il faut également indiquer l'utilisation du type générique dans le nom de la classe puisqu'elle sera instanciée de manière différente pour chaque type.

À retenir

Seule restriction, la déclaration de la classe et la définition des méthodes doivent se trouver dans le fichier d'en-tête, sinon il n'y a pas de compilation.

L'utilisation reste ensuite très classique, pour exemple, on fabrique une pile d'entiers et une pile de caractères. On empile 5 valeurs de chaque et on les dépile.

Classe Pile ***pile.cpp***

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

using namespace std;

int main()
{
   cout << "Voici une pile d'entiers" << endl;
   Pile<int> pile1(5);        // on précise que c'est une pile d'entiers

   for (int indice =0;indice < 5; indice++)
   {
       pile1.Empiler(indice);
   }
   for (int indice=0; indice < 5;indice++)
   {
        cout << pile1.Depiler() << endl;
   }
   cout << endl << "Voici une pile de caractères" << endl;
   Pile<char> pile2(5);              // on précise que c'est une pile de caractères

   for (int indice =0;indice < 5; indice++)
   {
       pile2.Empiler('A'+ static_cast<char>(indice));
   }
   for (int indice=0; indice < 5;indice++)
   {
        cout << pile2.Depiler() << endl;
   }
    return 0;
}
```

Les éléments sont bien affichés dans l'ordre inverse dans lequel ils ont été introduit dans la pile.

```
                                    Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Voici une pile d'entiers
4
3
2
1
0

Voici une pile de caractères
E
D
C
B
A
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

6. Programmation générique : Utilisation de la STL

6.1. Introduction

La librairie STL pour Standard Template Library, soit la librairie standard de modèle est une librairie normalisée pour le C++. Tous les identificateurs de la STL sont regroupés dans l'espace de nommage **std**. Cette librairie utilise massivement l'utilisation de types génériques développés lors du chapitre précédent.

Elle fournit un grand nombre d'éléments permettant une gestion efficace des structures de données complexes implémentées en C++, par exemple des tableaux, des piles, des files, des listes chaînées et des algorithmes optimisés pour les manipuler, recherche, tri, parcours, ajout ou encore suppression d'objets.

Plus particulièrement on y trouve trois familles d'éléments :

- **Les conteneurs** sont utilisés pour stocker des objets de même type. On parle donc de collection d'objets.
- **Les itérateurs** sont des objets qui permettent de naviguer parmi les éléments d'un conteneur. Un itérateur est un pointeur « intelligent ». L'itérateur fait le lien entre les conteneurs et les algorithmes
- **Les algorithmes** de la STL offrent des services fondamentaux sur les conteneurs. Ce sont des fonctions globales qui opèrent avec les itérateurs.

6.2. Les conteneurs

Les conteneurs sont des structures de données abstraites permettant de stocker des données de manière organisée. On en distingue deux catégories, les conteneurs séquentiels et les conteneurs associatifs. Ils diffèrent par la méthode d'accès aux données et à la façon dont la mémoire utilisée est organisée. Les premiers stockent les données de manière séquentielle, soit les une à la suite des autres. C'est le cas des vecteurs, des tableaux dynamiques, des piles, des files et des listes chaînées. Les seconds regroupent des ensembles ordonnés au sens mathématique ou des tables associatives ordonnées qui pourraient s'apparenter à des tableaux dont l'indexe ne serait pas forcément un entier, mais une chaîne de caractères par exemple, chaque donnée est associée à une clé permettant par la suite d'y accéder rapidement.

<table>
<tr><th colspan="2">Conteneurs séquentiels</th><th colspan="2">Conteneurs associatifs</th></tr>
<tr><th><u>Les tableaux</u></th><th><u>Les listes</u></th><th><u>Les ensembles ordonnés</u></th><th><u>Ensemble indexé par une clé</u></th></tr>
<tr><td>array
tableau statique contigu</td><td>forward_list
liste simplement chaînée</td><td colspan="2">N'accepte pas les doublons,
chaque donnée est unique dans la collection</td></tr>
<tr><td>vector
tableau dynamique contigu</td><td>list
liste doublement chaînée</td><td>set
collection triée</td><td>map
collection triée par une clé</td></tr>
<tr><td>Dérivés de vector</td><td></td><td>unordered_set
collection non triée</td><td>unordered_map
non triée avec clé</td></tr>
<tr><td>deque
file d'attente à 2 bouts</td><td></td><td colspan="2">Accepte les doublons</td></tr>
<tr><td>stack
pile (LIFO)</td><td></td><td>multiset
collection triée</td><td>mutimap
collection triée par une clé</td></tr>
<tr><td>queue
file d'attente (FIFO)</td><td></td><td>unordered_multiset
collection non triée</td><td>unordered_multimap
non triée avec clé</td></tr>
<tr><td>priority_queue
file d'attente avec priorité</td><td></td><td></td><td></td></tr>
</table>

Le détail des différents conteneurs ne peut pas être passé en revue dans ce document. En cas de besoin, il est nécessaire de ce reporté à la documentation de référence : https://en.cppreference.com/w/cpp/container ou à sa version française, dans une traduction approximative : https://fr.cppreference.com/w/cpp/container réaliser avec Google traduction. D'autres exemple sont présentés dans le cours sur les strutures et la gestion de données.

Le choix du type de conteneur dépend fondamentalement de l'utilisation et des performances que l'on souhaite en avoir par exemple, les conteneurs de type ***array***, ***vector, deque***, ***map et unorderd_map*** permettent d'accéder rapidement à un élément avec l'**opérateur [ ]** contrairement aux autres. Un élément peut être inséré n'importe où pour une ***list*** sans que cela soit pénalisant au niveau temps, à la fin ou au début pour un ***deque***, et uniquement à la fin pour un ***vector***.

Utiliser un ***vector***, est donc un bon choix lorsque l'on a besoin d'insérer/supprimer des éléments seulement à la fin, lorsque le conteneur doit être compatible au tableau C standard.

Utiliser une ***list*** est un bon choix lorsque l'on a besoin d'insérer/supprimer des éléments au milieu du conteneur et que le conteneur n'a pas besoin d'être compatible avec un tableau C standard ou que la taille maximum requise du conteneur n'est pas connue.

Le temps d'exécution pour les différentes opérations insertion, suppression, accès, recherche est fonction du type de conteneur et de sa structure. En fonction du lieu, la suppression ou l'insertion d'un élément dans un conteneur de type tableau peut demander la recopie complète de la collection alors que pour une liste c'est instantané. Par contre, l'accès à un élément sera instantané pour un tableau alors que pour une liste, elle doit être parcourue jusqu'à trouver l'élément.

Remarque

Les objets qui sont déposés dans un conteneur doivent répondre au modèle canonique dit **Coplien** et éventuellement avoir surchargés les opérateurs de tri comme inférieur **<** ou supérieur **>** et l'opérateur d'égalité **==.**

Pour de nombreux conteneurs, un ensemble de méthode est défini :	
empty()	Pour savoir si le conteneur est vide
size()	Pour déterminer le nombre d'éléments dans le conteneur
erase()	Pour supprimer un élément ou un ensemble d'éléments du conteneur
clear()	Pour supprimer tous les éléments

Pour déclarer un conteneur, comme pour chaque classe générique, il est nécessaire de préciser le type de données que contient le conteneur.

Exemple : ***conteneur.cpp***

```
#include <array>
#include <list>

class Rouleau;

using namespace std;

int main()
{
   // déclaration d'un tableau d'entiers avec 10 cases
   array <int, 10> tableau1;
   // déclaration d'un tableau de 3 caractères et son initialisation
   array <char, 3> tableau2 = {'a','b','c'};
   // déclaration d'une liste de Rouleaux, classe déclarée par ailleurs
   list <Rouleau>  listeDeRouleaux;


    return 0;
}
```

6.3. Les itérateurs

Un itérateur est un objet permettant manipuler plus facilement les éléments d'un conteneur. Il permet de le parcourir, d'accéder aux données et éventuellement de les modifier. Il existe deux types d'itérateurs, ceux qui permettent d'effectuer un parcours du début à la fin et ceux qui permettent l'inverse.

iterator	**Parcours du début à la fin**	***const_iterator***	L'élément désigné ne peut pas être modifié
reverse_iterator	**Parcours inverse**	***Const_reverse_iterator***	

Pour être utilisé, un itérateur doit être initialisé. Les méthodes **begin()** et **end()** présentes dans la plupart des conteneurs permettent une initialisation au début ou à la fin de la collection. D'autres méthodes retournant un conteneur comme pour une recherche **find()** réalise également son initialisation.

Pour sa déclaration, l'itérateur fait référence au conteneur auquel il va être associé et au type de données qu'il reçoit. Voici un exemple d'utilisation d'un conteneur de type ***vector*** avec un itérateur.

Exemple : ***conteneur_iterateur.cpp***

```
#include <vector>
#include <iostream>

using namespace std;

int main()
{
   //déclaration d'un tableau d'entiers dynamique avec 10 cases de réservées
   vector <int> tableau(10);

   for (int indice=0 ; indice < 10 ; indice++)
   {
        tableau[static_cast<size_t>(indice)] = indice;
   }

   //déclaration d'un itérateur sur le tableau
   vector<int>::iterator it;

   for (it=tableau.begin();it != tableau.end(); it++)
   {
       cout << *it << " ";
   }
   cout << endl;


    return 0;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
0 1 2 3 4 5 6 7 8 9
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

L'itérateur s'utilise comme un pointeur. Il a été initialisé avec la méthode **begin()** de ***vector***, le parcours se fait jusqu'à la fin du vecteur, sans se préoccuper des indices du tableau.

Remarque, l'avantage d'un tableau dynamique comme ***vector*** est qu'il peut s'agrandir ou se réduire en fonction des besoins.

Autre exemple avec un conteneur de type map :

Exemple : ***conteneur_iterateur.cpp***

```
#include <iostream>
#include <iomanip>
#include <map>


using namespace std ;
int
main ( void )
{
    map < string , string > telephones ;
    telephones ["Jean"]   = "0123456789";
    telephones ["Paul"]   = "0678912345";
    telephones ["Pierre"] = "0789123456";
    telephones ["Sophie"] = "0456789123";


    map < string , string >:: iterator it ;
    for ( it = telephones.begin() ; it != telephones.end() ; it ++)
    {
           cout << setw(7) << left << it->first << " -> " << it->second << endl ;
    }
    return 0;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Jean    -> 0123456789
Paul    -> 0678912345
Pierre  -> 0789123456
Sophie  -> 0456789123
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

it->first représente la clé,
it->second représente la valeur.

6.4. Les algorithmes de la librairie STL

La STL offre un ensemble d'algorithmes applicable aux différents conteneurs au travers des itérateurs. Parmi ces algorithmes, il existe plusieurs catégories :

Algorithmes de base `#include <algorithm>`	Recherche d'éléments : **min**, **max**, **find**, **search**... Déplacement d'éléments : **swap**, **move**, **reverse**, **rotate**... Tri : **sort**
Algorithmes numériques `#include <numeric>`	**accumulate** : effectue la somme des éléments
Autres algorithmes de tri `#include <cstdlib>`	**qsort** : Tri rapide, du plus petit au plus grand **qsort_r** : Tri rapide inverse

Les différents algorithmes sont présentés ici https://fr.cppreference.com/w/cpp/algorithm en français traduit approximativement par Google traduction, voici la version orignal : https://en.cppreference.com/w/cpp/algorithm.

Exemple : ***conteneur_tri.cpp***

```
#include <iostream>
#include <iomanip>
#include <array>
#include <algorithm>

using namespace std;

void AfficherValeur(int val);

int main()
{
    array <int, 10> tableau1 = {15,8,25,2,9,0,12,38,10,3};


    cout << "Tableau d'origine : ";
    for_each (tableau1.begin(), tableau1.end(),AfficherValeur);
    cout << endl;


    cout << "Tableau trié      : ";
    sort(tableau1.begin(),tableau1.end());
    for_each (tableau1.begin(), tableau1.end(),AfficherValeur);
    cout << endl;


    return 0;
}


void AfficherValeur(int val)
{
    cout << setw(5) << val;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Tableau d'origine :    15    8   25    2    9    0   12   38   10    3
Tableau trié      :     0    2    3    8    9   10   12   15   25   38
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Ce programme met en œuvre deux algorithmes de la librairie STL, **for_each** qui permet d'appeler une fonction pour chaque élément du conteneur et **sort** qui permet de le trier dans l'ordre croissant.

7. Programmation générique : patrons de développement

La programmation orientée-objet en C++ à permis d'introduire les concepts de surcharge, d'héritage, et de polymorphisme. Ces principes permettent d'accroire l'adaptabilité du code :

- en redéfinissant une méthode avec des paramètres différents tout en gardant un fonctionnement similaire,
- en spécialisant le code pour chaque membre d'une famille d'objets et ainsi obtenir une programmation générique.

Le chapitre sur les templates à permis de montrer encore une extension de ces concepts en les généralisant à différents types de données.

Les patrons de développement ou **design patterns** étendent cette idée de ré utilisabilité à la conception du logiciel. Ce mode de conception tend à rechercher des solutions standards qui pourraient s'appliquer à différents problèmes.

Lors de la phase de développement, les patrons servent de guide pour réaliser le code source du module en question.

Un patron regroupe un ensemble de composants apportant une solution à un problème. Ils sont regroupés par famille :

- Les créateurs : leur rôle est de définir les instanciations et la configuration des classes et des objets,
- Les structuraux : leur rôle est d'organiser les classes,
- Les comportementaux : ils décrivent la collaboration entre les objets, les algorithmes qui les composent et déterminent les responsabilités de chacun.

8. Conclusion

Le langage C++ est un langage polyvalent, réputé pour ses hautes performances, sa fiabilité, le faible encombrement du code produit et donc pour sa faible consommation d'énergie. Il permet de développer en bas niveau, au plus proche des composants ou en haut niveau, au niveau applicatif. C'est pourquoi on le retrouve dans le domaine de l'embarqué, du traitement d'image, des télécommunications, des jeux vidéo, des environnements graphiques, des applications scientifiques, dans l'écriture des systèmes d'exploitation... C'est dans les applications Web qu'il est peut-être le moins présent, même si côté serveur on peut le retrouver avec l'utilisation des applications utilisant les websocket par exemple.

Le langage C++ est l'un des langages les plus utilisés au monde. Il a la réputation d'être d'un abord plus complexe, mais ces performances et ces mécanismes de sûreté le rendent incontournable pour celui qui prend le temps de les appréhender. On retrouve sa syntaxe dans de nombreux autres langages. Certains d'entre eux sont même écrits en C++. Cela explique ces meilleures performances par rapport à ces autres langages. Connaître le C++ permet de s'adapter facilement à la programmation dans d'autres langages qui peuvent être dédiés à certains domaines.

C'est un langage en constante évolution, tout en gardant une compatibilité avec les versions antérieures, il est un gage de stabilité.

Le C++ est riche de nombreuses bibliothèques dans tous les domaines qui facilitent le développement d'application. Il peut-être complété par des frameworks tels que Qt qui le rende plus abordable et moderne.

Le C++ est le seul langage orienté objet qui est utilisé lors de l'épreuve écrite du BTS SN option Informatique et Réseaux.