Fiche n°12 : classes et objets (initiation au C++)

Nous entrons maintenant dans l'univers de la programmation objet et de C++. Une classe peut être vue comme une évolution d'une structure qui comporte non seulement des champs de données, appelés variables d'instance, mais aussi des constructeurs, des méthodes, et parfois un destructeur. Seuls quelques aspects du langage C++ seront abordés ici.

```cpp
class Date
{
  private:
  int jour;
  int mois;
  int annee;

  public:
  Date(int j, int m, int a);

  int compare(Date d);
  void print();
};
```

- Variables d'instances privées, accessibles seulement par les méthodes de la classe.
- Déclaration d'un constructeur.
- Déclaration de méthodes publiques, accessibles depuis toute l'application.

```cpp
Date::Date(int j, int m, int a)
{
    jour=j;
    mois=m;
    annee=a;
}
```

Définition du constructeur de la classe **Date**. Il permettra de créer des objets de type Date, aussi appelés **instances** de la classe Date.

```cpp
void Date::print()
{
    cout << jour << "/" << mois << "/" << annee;;
}
```

Cet opérateur (`<<`) envoie une valeur vers un flux, ici le flux **cout** qui représente la sortie standard, par défaut l'écran. **printf** est également utilisable.

Définition de la méthode print permettant l'affichage d'une date.

```cpp
int main()
{
    Date aujourdhui(9,2,2012);
    Date* hier = new Date(8,2,2012);
    aujourdhui.print();
    hier->print();

    return 0;
}
```

- La variable **aujourdhui** contient un objet de type Date.
- La variable **hier** reçoit un pointeur sur un objet Date créé dans une zone mémoire appelée tas.
- Appel de la méthode **print** avec l'objet **aujourdhui**.
- Appel de la méthode **print** avec l'objet pointé par **hier**.

Et la méthode Date::compare ? Elle reçoit en paramètre *la copie complète* d'un objet d de type Date qui doit être comparé à la date courante.

Si la date courante est située avant d, cette méthode doit retourner une valeur négative, si les deux dates sont identique, la valeur 0, sinon une valeur positive.

```cpp
int Date::compare(Date d)
{
    if(annee<d.annee) return -1;

    // a completer
}
```

Fiche n°13 : pointeurs, références, tableaux en C++

La méthode **compare** de la fiche précédente accepte en paramètre la copie complète d'un objet **Date**, ce qui n'est pas efficace en terme d'utilisation de la mémoire et de temps d'exécution.

Le problème peut être résolu avec le passage par pointeur.

```
int Date::compare(Date* d)
{
    if(annee < d->annee) return -1;

    // a completer

    return 0;
}
```

Une autre solution consiste à utiliser le **passage de paramètres par référence**.

Une référence est un alias qui permet l'accès au contenu d'une variable. Sa représentation en mémoire est aussi concise que celle d'un pointeur, mais il n'y a pas besoin d'utiliser l'opérateur de déréférencement * pour accéder au contenu de la variable concerné.

```
int Date::compare(Date& d)
{
    if(annee<d.annee) return -1;

    // a completer

    return 0;
}
```

La référence d permet l'accès en lecture et écriture à la variable passée en paramètre.

La référence d s'utilise comme une variable qui contient l'objet référencé.

Tableaux en C++

Les tableaux du langage C sont utilisables en C++. Il existe également un moyen de créer dynamiquement un tableau et de le détruire lorsqu'on en a plus besoin, pour libérer la mémoire occupée.

Attention : Pour que cette déclaration soit valide, il faut que la classe Date dispose d'un constructeur par défaut.

```
Date* t;

t = new Date[10];

t[3].print();

delete[] t;
```

```
Date::Date() {jour=1; mois=1; annee=2000;}
```

t[3] est un objet de type **Date**.

Destruction du tableau t

L'équivalence tableaux-pointeurs du langage C est aussi valable en C++.

Tableaux multidimensionnels

Les tableaux dynamiques à deux dimensions doivent être représentés sous la forme de tableaux de pointeurs sur des tableaux à une dimension.

L'approche est bien sûr généralisable au tableaux à plus de deux dimensions.

```
Date** t2;

t2 = new Date*[2];

t2[0] = new Date[3];
t2[1] = new Date[3];

t2[0][1].print();
```

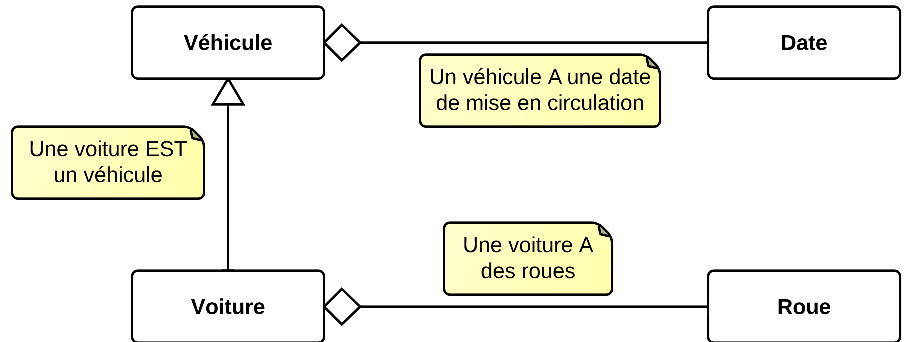
Tableau de 2 cellules contenant des pointeurs sur Date.

Tableau de 3 cellules contenant des objets de type Date.

Fiche n°14 : Héritage et composition

Le principe de la programmation orientée objet est très simple : une application est décomposée en **classes**. Chaque classe permet de créer des **objets**. Chaque objet contient des données et dispose de **méthodes**, qui remplacent les fonctions du langage C.

Il existe deux types de relations entre les classes : l'héritage et la composition.



L'héritage traduit le verbe être. Par exemple une voiture **est** un véhicule, donc la classe **Voiture** hérite, ou dérive, de la classe **Véhicule**. Toute instance, ou objet de la classe **Voiture** sera aussi considéré comme une instance de **Véhicule**.

La composition traduit le verbe avoir. Par exemple, une voiture **a** des roues, donc il y a un lien de composition entre la classe **Voiture** et la classe **Roue**. Toute instance de **Voiture** contiendra ou pointera ou fera référence à des instances de **Roue**.

Exemple simplifié d'application

Nous allons utiliser la classe standard **string** qui permet de gérer des chaînes de caractères. Cette classe utilise une représentation différence de celle du langage C.

Nous aurons besoin d'unefonction **i2s** pour convertir un entier en chaîne de caractères.

```
string i2s(int x)
{
    std::ostringstream flux;

    flux << x;

    return flux.str();
}
```

Ce flux standard convertit des informations en string.

On envoie l'entier x dans le flux.

La méthode str de la classe ostringstream retourne la chaîne produite par le flux.

Maintenant, ajoutons à la classe **Date** une méthode **toString** qui retourne une chaîne représentant la valeur d'un objet de type **Date**.

```
string Date::toString()
{
    return i2s(jour)+"/"+i2s(mois)+"/"+i2s(annee);
}
```

Dans ce contexte, l'opérateur + réalise la concaténation de chaînes.

Fiche n°15 : Héritage et composition (suite)

La classe **Vehicule** est définie de la manière suivante.

```
class Vehicule
{
    protected:
    int poids;
    Date miseEnCirculation;

    public:

    Vehicule(int p, Date m);

    int getPoids();
};
```

Ces deux **variables d'instance** sont **protected**, ce qui signifie que seules les méthodes de la classe **Vehicule** et des classes dérivées peuvent y accéder.

Ceci est la **déclaration** d'un constructeur de la classe **Vehicule**. Il accepte deux paramètres : le poids et la date de mise en service.

Ceci est la déclaration d'une méthode publique de la classe **Vehicule**. Elle permet d'obtenir le poids d'une instance de **Vehicule**.

```
Vehicule::Vehicule(int p, Date m)
{
    poids=p; miseEnCirculation=m;
}
```

```
int Vehicule::getPoids()
{
    return poids;
}
```

Ceci est la **définition** d'un constructeur de la classe **Vehicule**. Il initialise les variables d'instances.

Ceci est la définition de la méthode **getPoids**.

La classe **Voiture** est définie de la manière suivante.

```
class Voiture : Vehicule
{
    private:

    int nbPassagers;
    Roue tR[5];

    public:

    Voiture(int p, Date m, int nPass);
    string toString();
};
```

Deux variables d'instances pour représenter le nombre de places (un entier) et les 5 roues (un tableau d'instances de Roue).

Ceci *suppose* qu'une classe **Roue** a été définie.

Un constructeur et une méthode retournant une chaîne qui décrit la voiture.

```
Voiture::Voiture(int p, Date m, int nPass) : Vehicule(p,m)
{
    nbPassagers=nPass;
}
```

Ce constructeur appelle d'abord celui de la classe **Vehicule**.

```
string Voiture::toString()
{
    string s = "Voiture :\npoids : "+i2s(getPoids());
    s += "\nMise en circulation : "+miseEnCirculation.toString();
    s += "\nNombre de places : "+i2s(nbPassagers);
    return s;
}
```

Fiche n°16 : Héritage et composition (suite)

Le code d'une méthode peut être défini directement dans la classe qui la contient. Cette syntaxe doit être réservée aux méthodes ayant un code très concis. Par exemple, la classe **Roue** peut être définie de la manière suivante.

```
class Roue
{
    private:

    int usure;

    public:

    Roue() {usure=0;}

    void setUsure(int km) {usure=km;}

    int getUsure() {return usure;}
};
```

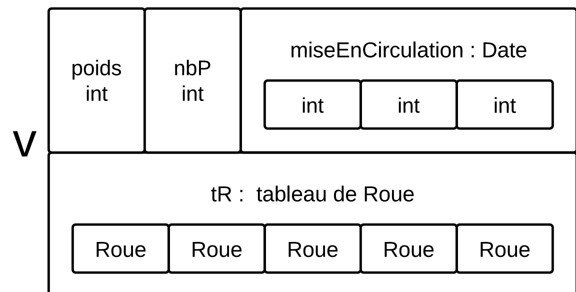
Constructeur par défaut (sans paramètre). Lors de la création d'un tableau d'instances de **Roue** (comme par exemple dans la classe **Voiture**), il est appelé pour initialiser chaque cellule.

Définitions complètes des deux « accesseurs » permettant de lire et de modifier la variable d'instance privée qui représente l'usure (en kms) d'une roue.

Voici un exemple de déclaration et d'utilisation d'une instance de la classe **Voiture**.

```
void testVoiture()
{
    Voiture v(702,Date(1,1,2012),5);

    cout << v.toString();
}
```



Lors de la création de la variable v, le constructeur de la classe **Voiture** est appelé.
Ce constructeur accepte en deuxième paramètre une instance de **Date** qui est ici fournie par un appel au constructeur de la classe **Date**.

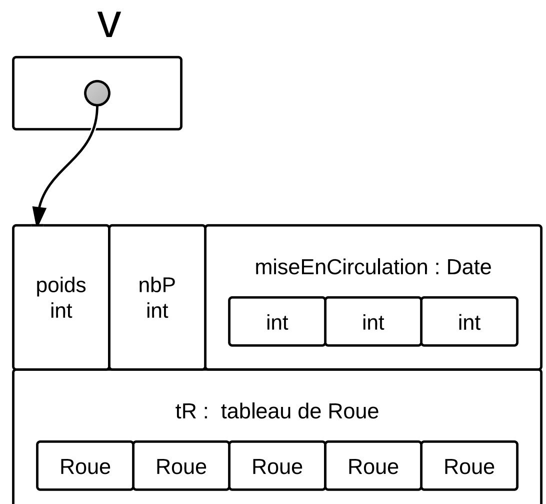
Dans cet autre exemple, l'instance de voiture est créée dynamiquement à l'aide de l'opérateur new.

```
void testVoiture()
{

 Voiture* v=new Voiture(702,Date(1,1,2012),5);

 cout << v->toString();

}
```



Fiche de préparation n°4

Quels sont les liens d’héritage et de composition entre les classes représentant une université, une personne, un étudiant, un enseignant et une UFR (ou faculté) ?

Réalisez l’implantation complète d’une classe PileInt qui représente une pile d’entiers et qui permet de créer une pile dont la taille est passée en paramètre au constructeur, d’empiler des entiers dans cette pile, de dépiler les valeurs empilées.