christophe.labourdette(at)cmla.ens-cachan.fr

Octobre 2016

Les classes

Une classe est un ensemble de données accompagnée d'une liste de méthodes possibles sur ces mêmes données.
Un objet de classe A est une région de la mémoire structurée selon la définition de la classe A. Il s'agit d'une instance de cette classe A.

```
class Individu {
    std::string nom, prenom;
    int age, poid, taille;
    int vieillit(int);
};
Individu paul;
paul.taille= 50; paul.age=0; paul.poid=3;
std::cout<<"Paul a maintenant ";
std::cout<<paul.vieillit(10) <<"\n";
```

Les fonctions membres

Il faut bien entendu déclarer la fonction membre :

```
int Individu::vieillit(int n)
{
  age += n;
  return age;
}
```

Vie et mort

vie

Un objet commence à vivre soit a partir de sa definition statique dans le code, soit lorsqu'il est cree dynamiquement.

mort

Un objet cesse de vivre soit quand on sort du contexte de sa definition (e.g. quand on sort d'une fonction ou l'objet est defini), soit lorsqu'il est detruit dynamiquement.

Constructeur et destructeur

Constructeur

Constructeur : méthode d'une classe qui est exécutée automatiquement quand un objet com- mence à vivre. Le constructeur porte le nom de la classe, peut avoir des arguments, mais pas de valeur de retour.

Destructeur

Destructeur : méthode d'une classe qui est exécutée automatiquement quand un objet cesse de vivre. Le destructeur porte le nom de la classe précédé de "$\sim$" et ne peut avoir ni arguments ni valeur de retour.

Dans le cas ou le programmeur n'en fournit pas le compilateur utilise un constructeur et un destructeur par défaut.

les références

En C++, une référence est un alias.

```
int bidule = 421;
int & truc = bidule; // truc est une reference a
      // bidule
truc += 5;                  // truc = 426 bidule = 426
```

La référence est sutout utilisée pour les passage de paramêtres dans les fonctions pour éviter un epassage par valeur.
La référence est un type de pointeur particulier qui ne peut changer de destination de pointage et doit être initialisée à la création.

Exemple

```
class Individu {
  . . .
public:
   Individu();
   ~Individu();
   Individu(int age, int taille, int poid);
   Individu(std::string nom, std::string prenom);
   Individu(int age, int taille, int poid,
            std::string nom, std::string prenom);
   . . .
};

Individu  machin;  //appel du constructeur
```

constructeurs

Il est parfaitement possible de définir (comme dans l'exemple précédent) de nombreux constructeurs différents, la condition est qu'ils doivent tous avoir une signature différente (nombre, type et ordre des argmuments).
Moralement, ce sont eux qui sont chargés de l'allocation de mémoire si nécessaire. Donc il peuvent contenir des *new*.
Le destructeur lui doit contenir les *delete* qui sont nécessaires, il est unique.

Constructeur par recopie

Il s'agit d'un constructeur qui prend en argument un élément de la classe et en recopie les données.

exemple

```
class Individu{
public:
Individu();
Individu(const Individu&);
. . .
};
 Individu A;
 Individu B(A);
```

Par défaut le compilateur crée un objet dont les membres prennent les mêmes valeurs que l'objet d'origine.

Problèmes avec la mémoire dynamique

Lors de l'utilisation d'un membre crée dynamiquement le constructeur par recopie par défaut va copier les adresses et non le contenu.
Deux objets différents vont alors avoir des valeurs membres au même endroit en mémoire. C'est rarement ce que l'on souhaite.
Par sécurité :**toujours définir un constructeur de recopie**.

Opérateur d'assignation

Lorsque l'on a :

```
Individu A,B;
. . .
A = B;
```

Le compilateur fait appel à l'opérateur d'assignation de la classe :

```
Individu & operator=(const Individu &);
```

Comme dans le cas du constructeur par recopie le compilateur fournit une méthode par défaut, mais elle pose les mêmes problèmes. Donc : **toujours définir un opérateur d'assignation**

Classe minimale ”orthodoxe”

Pour éviter des problèmes notamment lors de l'utilisation des classes et surtout lorsqu'elles comportent de la gestion dynamique une classe doit contenir au minimum :

Classe minimale

1. Un constructeur,
2. Un destructeur,
3. Un constructeur par recopie,
4. un opérateur d'assignation.

Les champs privés

Il existe un mot clé ”private” (par opposition à public).

public

Les champs définis dans l'espace *public* de la classe sont accessible pour tout le monde lors de l'utilisation de la classe.

private

Les champs définis dans l'espace *private* de la classe ne sont accessibles que par les méthodes membres de la classe.

Interface

Une utilisation classique et importante de l'espace private utilise la notion d'interface.

données privées

Par principe les données de la classe sont privées.
On souhaite que l'utilisateur ne puisse accéder aux données que par l'intermédiaire de méthodes d'interface.
On peut ainsi empêcher l'utilisateur de modifier des données, ne lui donner qu'un droit de lecture ou contrôler la façon dont il les modifie.

Il est également possible de réserver certaines méthodes pour des traitements internes à la classe.

Le mot clé friend

Le mot clé friend peut servir pour

- des fonctions
- des classes
- des fonctions membres

Il donne aux différents élements amis les mêmes accès que ceux des membres de la classe.
Il est particulièrement utilisé lors de la surcharge d'opérateurs, pour un opérateur binaire il se déclarera avec deux arguments mais aura accès aux données et fonctions privées de la classe.

this

Le pointeur this est un pointeur sur la classe courante.
Il permet de retourner des adresses et non plus des variables.
On le retrouve lui aussi lors de la surcharche d'opérateurs.

```
class truc {
public:
...
 truc& operator=(const truc& rhs)
 {
 ...
    return *this;
 }
 ...
}
```

operator

Il est possible en **C++** de surcharger quasiment tous les opérateurs, mais pas :

```
a.b
a.*b
a::b
a ? b : c
sizeof(a)
...
```

Les opérateurs =, $->$, (), [] , doivent être définis comme fonction membres.
En général, il est donc possible de choisir de définir l'opérateur, comme fonction membre ou fonction externe.

opérateur, membre ou externe ?

En prenant comme exemple l'addition comme fonction membre et la soustraction comme fonction externe on aura :

```
class truc{
  ...
   truc operator+(truc& B);
  ...
}
 truc truc::operator-(truc& A, truc& B);
```

const member

Il est parfois utile d'utiliser des fonctions membres constantes. Celles-ci ne peuvent modifier une donnée membre de la classe.

```
struct date {
int jour,mois,annee;
...
date(int,int,int);
date(std::string);
int jour() const; // ne peut pas changer jour
void jourplusun(); // incremente le jour
...
}
```