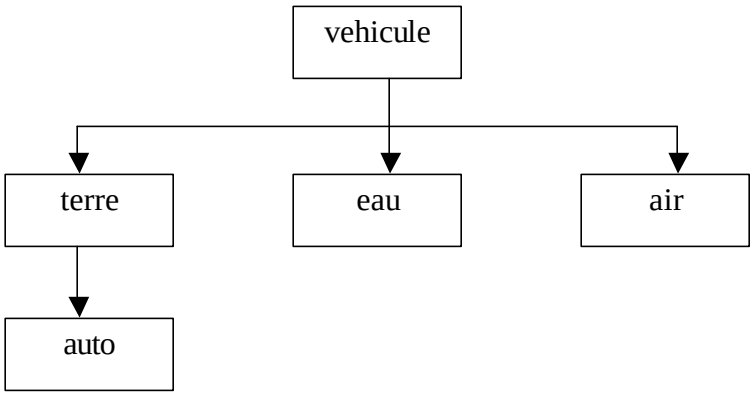
Chapitre 11 de “C++ Annotation” version 4.3.1, Frank B. Brokken et Karel Kubat.

Héritage

Classes reliées :



Définition de la classe vehicule :

```
class vehicule
{       public:
                //constructeurs
                vehicule();
                vehicule(int pd);
                //interface
                int getpoids() const;
                void setpoids(int pd);
        private:
                //données
                int poids;
};
```

la classe ‘terre’ est définie comme suit:

```
class terre
{       public :
                void setpoids(int pd);
        private:
                vehicule v;
};
void terre::setpoids(int pd)
{
        v.setpoids(pd);
}
```

Cette définition a deux problèmes :

1. Problème sémantique : la classe **terre** vehicule contient 'vehicule'. La relation correcte doit être : 'terre vehicule' est un cas spécial de 'vehicule'.

2. Problème pratique : un code non nécessaire est introduit. terre::setpoids fait seulement un appel à vehicule ::setpoids – on n'a pas ajouté une fonctionnalité extra, donc pourquoi ajoutons-nous un code extra ?

L'héritage résout ces 2 problèmes

```
{       public :
                //constructeurs
                terre();
                terre(int pd, int vi);
                //interface
                void setvitesse(int vi);
                int getvitesse() const;
        private :
                //données
                int vitesse;
};
```

La dérivation est définie par post-fixer le nom de la classe 'terre', dans sa définition, par 'public vehicule' : la classe 'terre' contient maintenant toute la fonctionnalité de sa classe de base 'vehicule' plus ses propres informations.

Exemple d'utilisation de la classe dérivée :

```
terre veh(1200,145);

int main()
{       cout << "poids du vehicule " << veh.getpoids() << endl
                << "La vitesse est" << veh.getvitesse() << endl;
        return(0);
}
```

Cet exemple montre deux des caractéristiques de la dérivation :

1. getpoids() n'est pas un membre direct de terre. Cependant elle est utilisée dans veh.getpoids() – cette fonction membre est une partie implicite de la classe, elle est héritée de sa classe parent 'vehicule'.

2. Les champs privés (private) de 'vehicule' demeurent privés, même si la classe dérivée 'terre' contient la fonctionnalité de la classe parent 'vehicule' : ils sont seulement accessibles à partir des fonctions membres de 'vehicule' - 'terre' doit utiliser les fonctions getpoids et setpoids de 'vehicule' pour adresser le champ 'poids', juste comme n'importe quel code dehors la classe 'vehicule'.

De la même façon, on peut crée une classe 'auto' dérivée de 'terre' – ceci est appelé dérivation emboîtée.

Exemple :

```
class auto : public terre
{       public :
                //constructeurs
                auto();
                auto(int pd, int vi, char const *nm);
                /interface
                char const *getnom() const;
                void setnom(char const *nm);
        private:
                //données
                char const *nom;
};
```

'auto' contient le poids, la vitesse et le nom de l'automobile.

Le constructeur de la classe dérivée

Le constructeur de 'terre' peut être définit comme suit :

```
terre::terre(int pd, int vi)
{
        setpoids(pd);
        setvitesse(vi);
};
```

-Ce n'est pas efficace car setpoids doit faire un appel à vehicule ::setpoids. La méthode la plus efficace est d'appeler directement le constructeur de 'vehicule' :

```
terre::terre(int pd, int vi) : vehicule(pd)
{
        setvitesse(vi);
};
```

Redéfinir les fonctions membres

On peut redéfinir les actions de toutes les fonctions définies dans une classe base.

Supposons que camion est représenté par 2 parties : le moteur et la carrosserie – tous les deux parties ont leurs propres poids, mais la fonction ‘getpoids‘ doit retourner la combinaison des deux poids.

Exemple :

```
Class camion :public auto
{       public:
                //constructeurs
                camion();
                camion(int pd_moteur, int vi, char const *nm, int pd_carrosserie);
                //interface : pour définir 2 champs de poids
                void detpoids(int pd_moteur, int pd_carrosserie);
                //et retourne poids combinés
                int getpoids() const;

        private:
                //données
                //le poids du  moteur est représenté dans la classe ‘auto’
                int poids_carrosserie;
};

//constructeur
camion ::camion(int pd_moteur, int vi, char const *nm, int pd_carrosserie)
:auto(pd_moteur,vi,nm)
{
        poids_carrosserie = pd_carrosserie;
};
```

On redéfinit les fonctions setpoids et getpoids comme suit :

```
void camion::setpoids(int pd_moteur, int pd_carrosserie)
{
        poids_carrosserie = pd_carrosserie;
        setpoids(pd_moteur); // utilise auto::setpoids()
};

int camion::getpoids() const
{
        return(
                auto::getpoids() +      //somme de la partie moteur
                poids_carrosserie);     //et la carrosserie
};
```

Noter que auto::getpoids() doit être 'void' sinon on peut avoir une récursivité infinie.

Héritage Multiple

Il est possible qu'une classe soit dérivée de plusieurs classes bases – une telle classe hérite sa fonctionnalité de plusieurs 'parent' en même temps.

Exemple : une classe 'moteur' peut stocker des informations sur le moteur. (numéro séquentiel, puissance, type d'essence) peuvent être définies.

```
class moteur
{       public :
                …       //constructeurs et interface
        private :
                //données
                char const *numero_seq, *type_essence;
                int puissance;
};
```

pour représenter une auto avec les informations extra du 'moteur', on peut définir une nouvelle classe, 'moteurauto', dérivée de 'auto' et de 'moteur' simultanément :

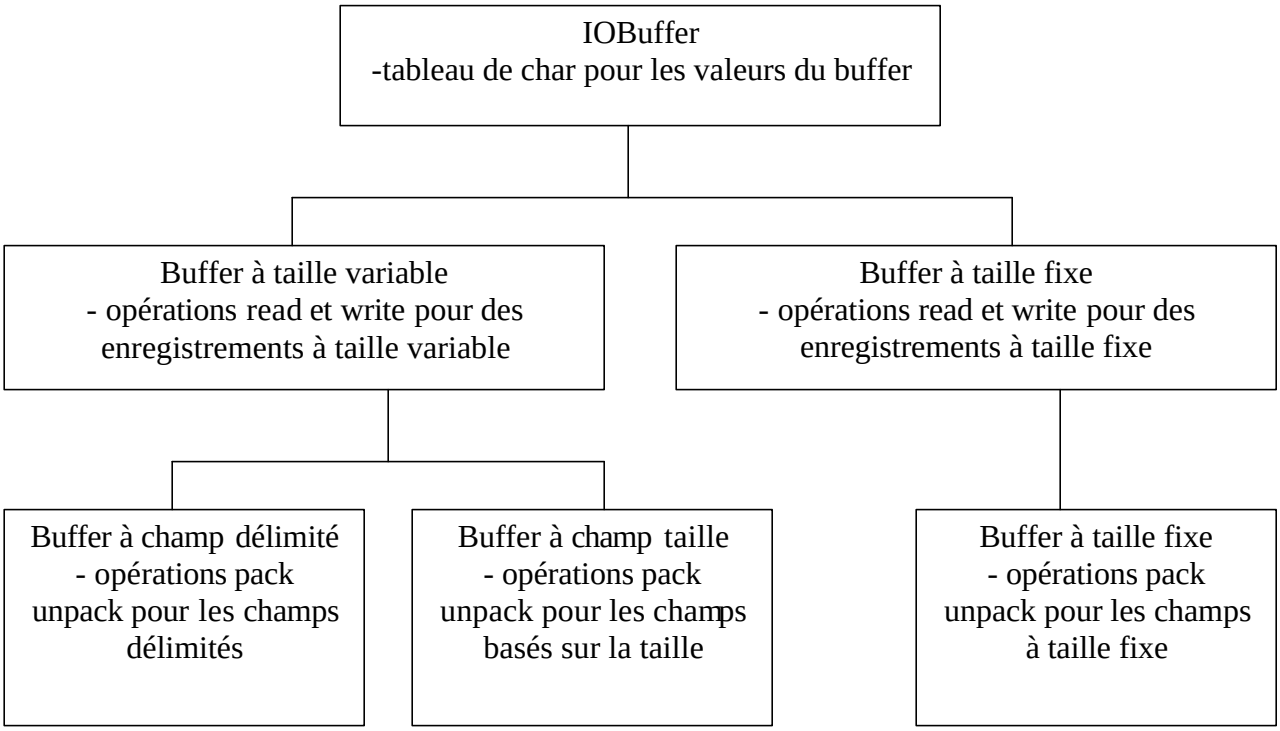
```
class moteurauto : public auto, public moteur
{       public:
                //constructeurs
                moteurauto();
                moteurauto(int pd, int vi, char const *nm, char const *numseq, int pui,
                char const *essence);
};

moteurauto::moteurauto(int pd, int vi, char const *nm, char const *numseq, int pui, char
const *essence);
{
                moteur(numseq,pui,essence);
                auto(pd,vi,nm);
};
```

-Note : du coté sémantique, cette définition est un peu bizarre car elle suggère que ‘moteurauto’ soit un auto et un moteur au lieu de dire qu’un ‘moteurauto’ possède un moteur – mais si on applique cette définition on va avoir un code dupliqué.

Un exemple sur l’importance de l’héritage pour la structure des fichiers (Section 4.2-5 dans “File Structures : An Object-Oriented Approach with C++” Folk, Zoellick et Riccardi)

C’est mieux de stocker un enregistrement dans une mémoire tampon avant de l’écrire dans un fichier – c’est particulièrement utile pour la représentation des tailles variables, tout en mettant un indicateur de taille au début de chaque enregistrement : la taille peut être obtenue au moment du stockage de l’enregistrement dans la mémoire tampon – puis cette taille va être écrite dans le fichier suivie par le contenu de la mémoire tampon.

Il existe une organisation hiérarchique naturelle pour les différents types de mémoire tampon qui peuvent être implémentées à travers les outils d'héritage de C++ - la hiérarchie est représentée comme suit :




```
Class IOBuffer
{       public :
                IOBuffer(int maxBytes = 1000);
                virtual int read(istream &)=0;
                virtual int write(ostream &) const=0;
                virtual int pack(const void *field, int size = -1) =0;
                virtual int unpack(void *field, int maxbytes=-1)=0;
        protected :
                char * buffer;
                int BufferSize;
                int MaxBytes;
}
```

Notes:

a. Les méthodes avec le mot-clé **virtual** sont des méthodes abstrait – chaque classe va définir sa propre implémentation.

b. Le mot-clé **virtual** est utilisé dans le cas d'héritage multiple pour assurer que les classes communes à 2 différents antécédents sont inclut une seule fois dans l'ascendance de la classe qui hérite.

c. Les éléments dans '**protected**' sont visibles aux classes dérivées mais pas aux autres classes.

Presque tous les membres et les méthodes de la classe **buffer** sont identiques. Les seules différences sont dans le packing et unpacking, et dans les différences minimes dans 'read' et 'write' entre les structures d'enregistrement à taille fixe et à taille variable.

-Beaucoup de code va être partagé en cas d'héritage :
Exemple : les enregistrements délimités et les enregistrements basés sur la taille (avec une taille variable) , peuvent tous les deux utiliser les mêmes méthodes 'read' et 'write' propres aux enregistrements à taille variable.