Chapitre 3

HERITAGE

3.1 Relations *a-un*, *est-un*, *utilise-un*

(3.1.1) Dans le cadre d'un programme concernant les transports, supposons que nous déclarions les quatre classes suivantes : `Voiture`, `Moteur`, `Route` et `Vehicule`. Quel genre de relation y a-t-il entre la classe `Voiture` et les trois autres classes ?

Tout d'abord, on peut dire qu'une voiture *a un* moteur : le moteur fait partie de la voiture, il est contenu dans celle-ci.

Ensuite, une voiture *utilise une* route, mais il n'y a pas d'inclusion : la route ne fait pas partie de la voiture, de même que la voiture ne fait pas partie de la route.

Enfin, une voiture *est un* véhicule, d'un genre particulier : la voiture possède toutes les caractéristiques d'un véhicule, plus certaines caractéristiques qui lui sont propres.

(3.1.2) Du point de vue de la programmation, la relation *a-un* est une inclusion entre classes. Ainsi, un objet de type `Voiture` renferme une donnée-membre qui est un objet de type `Moteur`.

La relation *utilise-un* est une collaboration entre classes indépendantes. Elle se traduit le plus souvent par des pointeurs. Par exemple, un objet de type `Voiture` renfermera une donnée-membre de type pointeur sur `Route`, ce qui permettra à la voiture de communiquer avec une route (cf. (2.6.3)).

La relation *est-un* s'appelle un *héritage* : une voiture hérite des caractéristiques communes à tout véhicule. On dira que la classe `Voiture` est *dérivée* de la classe `Vehicule`.

3.2 Classes dérivées

(3.2.1) Le principe est d'utiliser la déclaration d'une classe — appelée *classe de base* ou classe *parente* — comme base pour déclarer une seconde classe — appelée *classe dérivée*. La classe dérivée héritera de tous les membres (données et fonctions) de la classe de base.

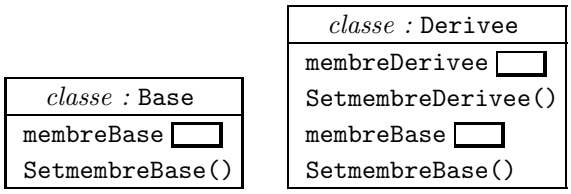
Considérons par exemple la déclaration suivante :

```
class Base
{
public:
        short membreBase;
        void SetmembreBase(short valeurBase);
};
```

On déclare une classe dérivée de la classe `Base` grâce au qualificatif `public Base`. Par exemple :

```
class Derivee :  public Base          // héritage public
{
public:
        short membreDerivee;
        void SetmembreDerivee(short valeurDerivee);
};
```

Un objet de la classe `Derivee` possède alors ses propres données et fonctions-membres, plus les données-membres et fonctions-membres héritées de la classe `Base` :



(3.2.2) **Contrôle des accès**

Il est possible de réserver l'accès à certaines données (ou fonctions) membres de la classe de base aux seules classes dérivées en leur mettant le qualificatif `protected:`

A retenir :

Les données et fonctions-membres privées sont inaccessibles aux classes dérivées.

(3.2.3) **Premier exemple**

```
# include <iostream.h>

class Base
{
public:
        void SetmembreBase(short valeurBase);
protected:
        short membreBase;
};

void Base::SetmembreBase(short valeurBase)
{
        membreBase = valeurBase;
}

class Derivee :  public Base
{
public:
        void SetmembreDerivee(short valeurDerivee);
        void AfficheDonneesMembres(void);
private:
        short membreDerivee;
};

void Derivee::SetmembreDerivee(short valeurDerivee)
{
        membreDerivee = valeurDerivee;
}

void Derivee::AfficheDonneesMembres(void)
{
        cout << "Le membre de Base a la valeur " << membreBase << "\n";
        cout << "Le membre de Derivee a la valeur " << membreDerivee << "\n";
}

void main()
{
        Derivee *ptrDerivee;
        ptrDerivee = new Derivee;
        ptrDerivee -> SetmembreBase(10);              // message 1
        ptrDerivee -> SetmembreDerivee(20);           // message 2
        ptrDerivee -> AfficheDonneesMembres();        // message 3
}
```

A l'exécution, ce programme affichera les deux lignes suivantes :

```
Le membre de Base a la valeur 10
Le membre de Derivee a la valeur 20
```

(3.2.4) **Héritage public ou privé**

Il est possible de déclarer :

```
        class Derivee :  public Base          // héritage public
ou :    class Derivee :  private Base         // héritage privÝ
```

Il faut savoir que :

- *dans le premier cas, les membres hérités conservent les mêmes droits d'accès (*`public` *ou* `protected`*) que dans la classe de base,*
- *dans le second cas (cas par défaut si rien n'est précisé), tous les membres hérités deviennent privés dans la classe dérivée.*

On conseille généralement d'utiliser l'héritage public, car dans le cas contraire on se prive de pouvoir créer de nouvelles classes elles-mêmes dérivées de la classe dérivée.

(3.2.5) **Constructeurs et destructeurs**

Quand un objet est créé, si cet objet appartient à une classe dérivée, le constructeur de la classe parente est *d'abord* appelé. Quand un objet est détruit, si cet objet appartient à une classe dérivée, le destructeur de la classe parente est appelé *après*.

Ces mécanismes se généralisent à une chaîne d'héritages. Voici un exemple :

```
# include <iostream.h>

class GrandPere
{
        .....  // données-membres
public:
        GrandPere(void);
        ~GrandPere();
};

class Pere :  public GrandPere
{
        .....  // données-membres
public:
        Pere(void);
        ~Pere();
};

class Fils :  public Pere
{
        .....  // données-membres
public:
        Fils(void);
        ~Fils();
};

void main()
{
        Fils *junior;
        junior = new Fils;
                // appels successifs des constructeurs de GrandPere, Pere et Fils
        ........
        delete junior;
                // appels successifs des destructeurs de Fils, Pere et GrandPere
}
```

(3.2.6) **Cas des constructeurs paramétrés**

Supposons déclarées les classes suivantes :

```
class Base
{
        .....
        Base(short val);
};

class Derivee :  public Base
{
        .....
        Derivee(float x);
};
```

Dans la définition du constructeur de `Derivee`, on pourra indiquer quelle valeur passer au constructeur de la classe `Base` de la manière suivante :

```
Derivee::Derivee(float x) :  Base(20)
{
        .....
}
```

(3.2.7) **Deuxième exemple**

```
# include <iostream.h>

class Rectangle
{
public:
        Rectangle(short l, short h);
        void AfficheAire(void);
protected:
        short largeur, hauteur;
};

Rectangle::Rectangle(short l, short h)
{
        largeur = l;
        hauteur = h;
}

void Rectangle::AfficheAire()
{
        cout << "Aire = " << largeur * hauteur << "\n";
}

class Carre :  public Rectangle
{
public:
        Carre(short cote);
};

Carre::Carre(short cote) :  Rectangle(cote, cote)
{
}

void main()
{
        Carre *monCarre;
        Rectangle *monRectangle;
        monCarre = new Carre(10);
        monCarre -> AfficheAire();                    // affiche 100
        monRectangle = new Rectangle(10, 15);
        monRectangle -> AfficheAire();                // affiche 150
}
```

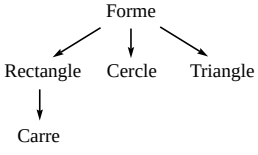
Nous retiendrons ceci :

Grâce à l'héritage, avec peu de lignes de code on peut créer de nouvelles classes à partir de classes existantes, sans avoir à modifier ni recompiler ces dernières.

Le génie logiciel fait souvent usage de librairies toutes faites, contenant des classes qu'il suffit de dériver pour les adapter à ses propres besoins.

3.3 Polymorphisme

(3.3.1) Dans l'exemple précédent, `AfficheAire()` de `Carre` réutilise le code de `AfficheAire()` de `Rectangle`. Mais dans d'autres cas, il peut être nécessaire d'écrire un code différent. Par exemple, dans une hiérarchie de classes de ce genre :



on a une version de `AfficheAire()` pour chacune de ces classes.

Si ensuite on crée une collection d'objets de type `Forme`, en demandant `AfficheAire()` pour chacune de ces formes, ce sera automatiquement la version correspondant à chaque forme qui sera appelée et exécutée : on dit que `AfficheAire()` est *polymorphe*. Ce choix de la version adéquate de `AfficheAire()` sera réalisé au moment de l'exécution.

Toute fonction-membre de la classe de base devant être surchargée (c'est-à-dire redéfinie) dans une classe dérivée doit être précédée du mot `virtual`.

(3.3.2) **Exemple**

```
# include <iostream.h>

class Forme
{
        // données et fonctions-membres....
public:
        virtual void QuiSuisJe(void);  // fonction destinée à être surchargée
};

void Forme::QuiSuisJe()
{
        cout << "Je ne sais pas quel type de forme je suis !\n";
}

class Rectangle :  public Forme
{
        // données et fonctions-membres....
public:
        void QuiSuisJe(void);
};

void Rectangle::QuiSuisJe()
{
        cout << "Je suis un rectangle !\n";
}

class Triangle :  public Forme
{
        // données et fonctions-membres....
public:
        void QuiSuisJe(void);
};

void Triangle::QuiSuisJe()
{
        cout << "Je suis un triangle !\n";
}

void main()
{
        Forme *s;
        char c;
        cout << "Voulez-vous créer  1 : un rectangle ?\n";
        cout << "                   2 : un triangle ?\n";
        cout << "                   3 : une forme quelconque ?\n";
        cin >> c;
        switch (c)
        {
                case '1' :  s = new Rectangle; break;
                case '2' :  s = new Triangle; break;
                case '3' :  s = new Forme;
        }
        s -> QuiSuisJe();      // (*) cet appel est polymorphe
}
```

Remarque.— Pour le compilateur, à l'instruction marquée `(*)`, il est impossible de savoir quelle version de `QuiSuisJe()` il faut appeler : cela dépend de la nature de la forme créée, donc le choix ne pourra être fait qu'au moment de l'exécution (on appelle cela *choix différé*, ou *late binding* en anglais).

(3.3.3) *Remarques :*

- Une fonction déclarée virtuelle doit être définie, même si elle ne comporte pas d'instruction.
- Un constructeur ne peut pas être virtuel. Un destructeur peut l'être.

(3.3.4) **Compatibilité de types**

Supposons déclaré :

```
class A
{
        .....
};

class B : public A      // ou private A
{
        .....
};

A a;
B b;
A *pa;
B *pb;
void *pv;
```

Alors :

- l'affectation `a = b` est correcte ; elle convertit automatiquement `b` en un objet de type `A` et affecte le résultat à `a`
- l'affectation inverse `b = a` est illégale
- de la même manière, l'affectation `pa = pb` est correcte
- l'affectation inverse `pb = pa` est illégale ; on peut cependant la forcer par l'*opérateur de conversion de type* `()`, en écrivant `pb = (B*) pa`
- l'affectation `pv = pa` est correcte, comme d'ailleurs `pv = pb`
- l'affectation `pa = pv` est illégale, mais peut être forcée par `pa = (A*) pv`.

(3.3.5) **Opérateur de portée `::`**

Lorsqu'une fonction-membre virtuelle `f` d'une classe `A` est surchargée dans une classe `B` dérivée de `A`, un objet `b` de la classe `B` peut faire appel aux deux versions de `f` : la version définie dans la classe `B` — elle s'écrit simplement `f` — ou la version définie dans la classe parente `A` — elle s'écrit alors `A::f`, où `::` est l'*opérateur de portée*.

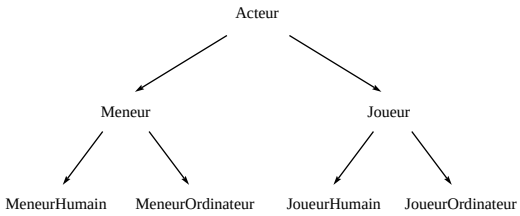
3.4 Exemple

(3.4.1) Reprenons le programme de jeu "c'est plus, c'est moins" du chapitre 2, §5. Nous pouvons maintenant écrire une version dans laquelle, au moment de l'exécution, les rôles du joueur et du meneur seront attribués soit à un humain, soit à l'ordinateur. L'ordinateur pourra donc jouer contre lui-même ! Il est important de noter que le déroulement de la partie, qui était contrôlé par la boucle :

```
do                          // voir la fonction main(), paragraphe (2.5.7)
{
        p = j.Propose(d);   // proposition du joueur
        d = m.Reponds(p);   // diagnostic du meneur
}
while (d && cpt < 6);
```

s'écrira de la même façon dans cette nouvelle version, quelle que soit l'attribution des rôles. Cela est possible grâce au polymorphisme des fonctions `Propose()` et `Reponds()`.

Nous utiliserons la hiérarchie de classes suivante :



(3.4.2) Conformément à (1.4.2), nous écrirons un fichier de déclaration `.h` et un fichier de définition `.cpp` pour chacune de ces sept classes. Afin d'éviter que, par le jeu des directives d'inclusion `#include`, certains fichiers de déclarations ne soient inclus plusieurs fois (ce qui provoquerait une erreur à la compilation), nous donnerons aux fichiers `.h` la structure suivante :

```
#ifndef SYMBOLE                     // si SYMBOLE n'est pas défini ...
#define SYMBOLE                     // ...  définir SYMBOLE
....  // ici, les déclarations normalement prévues
#endif                              // fin du si
```

`#ifndef` est une *directive de compilation conditionnelle* : elle signifie que les lignes qui suivent, jusqu'au `#endif`, ne doivent être compilées que si `SYMBOLE` n'est pas déjà défini. Or, en vertu de la deuxième ligne, ceci n'arrive que lorsque le compilateur rencontre le fichier pour la première fois.

(3.4.3) **Déclarations des classes**

```
//---------------------- acteur.h ----------------------

#ifndef _ACTEUR_H
#define _ACTEUR_H

class Acteur
{
public:
       void AfficheNom();
protected:
       char nom[20];
};

#endif


//---------------------- meneur.h ----------------------

#ifndef _MENEUR_H
#define _MENEUR_H

#include "acteur.h"

class Meneur :  public Acteur
{
public:
       virtual int Reponds(int prop);
};

#endif


//---------------------- joueur.h ----------------------

#ifndef _JOUEUR_H
#define _JOUEUR_H

#include "acteur.h"

class Joueur :  public Acteur
{
public:
       virtual int Propose(int diag);
};

#endif
```

```
//---------------------- meneurhu.h ----------------------

#ifndef _MENEURHU_H
#define _MENEURHU_H

#include "meneur.h"

class MeneurHumain :  public Meneur
{
public:
        MeneurHumain();
        int Reponds(int prop);
};

#endif
```

```
//---------------------- meneuror.h ----------------------

#ifndef _MENEUROR_H
#define _MENEUROR_H

#include "meneur.h"

class MeneurOrdinateur :  public Meneur
{
public:
        MeneurOrdinateur();
        int Reponds(int prop);
private:
        int numsecret;
};

#endif
```

```
//---------------------- joueurhu.h ----------------------

#ifndef _JOUEURHU_H
#define _JOUEURHU_H

#include "joueur.h"

class JoueurHumain :  public Joueur
{
public:
        JoueurHumain();
        int Propose(int diag);
};

#endif
```

```
//---------------------- joueuror.h ----------------------

#ifndef _JOUEUROR_H
#define _JOUEUROR_H

#include "joueur.h"

class JoueurOrdinateur :  public Joueur
{
public:
        JoueurOrdinateur();
        int Propose(int diag);
private:
        int min, max, proposition;
};

#endif
```

(3.4.4) **Définitions des classes**

```
//---------------------- acteur.cpp ----------------------

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

```
void Acteur::AfficheNom()
{
        cout << nom;
}



//---------------------- meneur.cpp ----------------------

#include "meneur.h"

int Meneur::Reponds(int prop)
{
        // rien à ce niveau
}

//---------------------- joueur.cpp ----------------------

#include "joueur.h"

int Joueur::Propose(int diag)
{
        // rien à ce niveau
}

//---------------------- meneurhu.cpp ----------------------

#include "meneurhu.h"
#include <iostream.h>

MeneurHumain::MeneurHumain()
{
        cout << "Vous faites le meneur.  Quel est votre nom ?  ";
        cin >> nom;
        cout << "Choisissez un numéro secret entre 1 et 100.\n\n";
}

int MeneurHumain::Reponds(int prop)
{
        AfficheNom();
        cout << ", indiquez votre réponse :";
        int r;
        cout << "\n0 - C'est exact";
        cout << "\n1 - C'est plus";
        cout << "\n2 - C'est moins";
        cout << "\nVotre choix (0, 1 ou 2) ?  ";
        cin >> r;
        return r;
}

//---------------------- meneuror.cpp ----------------------

#include "meneuror.h"
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

MeneurOrdinateur::MeneurOrdinateur()
{
        strcpy(nom, "l'ordinateur");
        srand((unsigned) time(NULL));
        numsecret = 1 + rand() % 100;
}

int MeneurOrdinateur::Reponds(int prop)
{
        cout << "Réponse de ";
        AfficheNom();
        if (prop < numsecret)
        {
                cout << " :  c'est plus\n";
                return 1;
        }
        else if (prop > numsecret)
        {
                cout << " :  c'est moins\n";
                return 2;
```

```
        }
        cout << " :  c'est exact\n";
        return 0;
}

//---------------------- joueurhu.cpp ----------------------

#include "joueurhu.h"
#include <iostream.h>

JoueurHumain::JoueurHumain()
{
        cout << "Vous faites le joueur.  Quel est votre nom ?  ";
        cin >> nom;
}

int JoueurHumain::Propose(int diag)
{
        AfficheNom();
        int p;
        cout << ", votre proposition ?  ";
        cin >> p;
        return p;
}

//---------------------- joueuror.cpp ----------------------

#include "meneuror.h"
#include <iostream.h>
#include <string.h>

JoueurOrdinateur::JoueurOrdinateur()
{
        strcpy(nom, "l'ordinateur");
        min = 1;
        max = 100;
        proposition = 0;
}

int JoueurOrdinateur::Propose(int diag)
{
        if (diag == 1)
                min = proposition + 1;
        else
                max = proposition - 1;
        proposition = (min + max) / 2;
        AfficheNom();
        cout << " propose :  " << proposition << "\n";
        return proposition;
}
```

(3.4.5) **Traitement principal**

```
//---------------------- jeu.cpp ----------------------

#include "meneur.h"
#include "meneurhu.h"
#include "meneuror.h"
#include "joueur.h"
#include "joueurhu.h"
#include "joueuror.h"
#include <iostream.h>

void main()
{
        cout << "\n\n\tJEU DU C++, C--\n\n";
        Joueur *j;
        Meneur *m;

                        // distribution des rôles

        char rep;
        cout << "Qui est le meneur (h = humain, o = ordinateur) ?  ";
        cin >> rep;
        if (rep == 'h')
                m = new MeneurHumain;
        else
```

```
        m = new MeneurOrdinateur;
    cout << "Qui est le joueur (h = humain, o = ordinateur) ?  ";
    cin >> rep;
    if (rep == 'h')
        j = new JoueurHumain;
    else
        j = new JoueurOrdinateur;

            // déroulement de la partie

    int p, d = 1, cpt = 0;
    do
    {
        p = j -> Propose(d);
        d = m -> Reponds(p);
        cpt++;
    }
    while (d && cpt < 6);

            // affichage du résultat

    cout << "\nVainqueur :  ";
    if (d)
        m -> AfficheNom();
    else
        j -> AfficheNom();
}
```