Chapitre 4 : Héritage de classes, fonctions virtuelles et classes abstraites

Table des matières

1 Héritage

L'héritage permet de réutiliser des classes existantes en les enrichissant de fonctions et champs supplémentaires pour un usage plus précis. Lorsque l'on pense définir de nouvelles classes, il est important de ne pas réinventer systématiquement la roue mais plutôt de bien réfléchir à l'intégration de la nouvelle classe au paysage existant. Cela demande en particulier d'avoir une idée claire *a priori* de ce que l'on pense faire et également d'avoir une bonne connaissance des classes existantes.

1.1 Héritage simple

Le mécanisme de l'héritage simple consiste en la définition d'une classe `ma_classe_derivee` par extension d'une classe `ma_classe_de_base` : la donnée d'un objet de `ma_classe_derivee` est la donnée d'un objet de `ma_classe_de_base` PLUS d'autres membres ou méthodes propres à `ma_classe_derivee`. Une autre façon de voir est de dire que les objets de la classe dérivée ont, de façon implicite, en plus de leurs membres et méthodes propres, un membre qui serait un objet de la classe de base et les méthodes de la classe de base (qui s'appliquent, *a priori*, à ce membre "supplémentaire").

La syntaxe est la suivante.

`class ma_classe_derivee : public` ou `private` ou `protected : ma_classe_de_base` { déclaration } ;

Exemple :

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

class point { public : int abs, ord;
        point(int x=0, int y=0) : abs(x), ord(y) {};};

class pixel : public point {
        public : string couleur;
        pixel(int x, int y, string c) : point(x,y), couleur(c) {};
        void affiche() {cout<<abs<<" "<<ord<<" "<<couleur<<endl;};};

int main() {pixel p(5,68,"rouge");
        p.affiche();}
```

(nom du fichier : `Chapitre4/prog01.cpp`*)*

Le choix du mot clé `public` ou `private` ou `protected` détermine les **possibilités d'accès**, pour un objet de la classe dérivée, aux membres et méthodes de l'objet de la classe de base qui lui est sous-jacent.

Dans un contexte d'héritage (qui peut être transitif, une classe dérivée pouvant être elle-même la classe de base d'une autre classe dérivée) il existe 4 genres de membres et méthodes :

— les membres et méthodes `public`,
— les membres et méthodes `private`,
— les membres et méthodes `protected` (absents jusque là car, hors héritage, ils sont équivalents aux `private`),
— les membres et méthodes inaccessibles.

Seuls les membres `public` sont accessibles dans une fonction extérieure.

Les tableaux suivants permettent de comprendre comment ces 4 genres se comportent vis-à-vis des 3 types d'héritages (`public`, `private` ou `protected`).

Héritage `public`	
Membres de la classe de base	Statut dans la classe dérivée
publics	publics
protégés	protégés
privés	inaccessibles
inaccessibles	

Héritage `protected`	
Membres de la classe de base	Statut dans la classe dérivée
publics	protégés
protégés	
privés	inaccessibles
inaccessibles	

Héritage `privé`	
Membres de la classe de base	Statut dans la classe dérivée
publics	privés
protégés	
privés	inaccessibles
inaccessibles	

Par exemple, lors d'un héritage `private`, les membres publics de l'objet de la classe de base deviennent privés, donc inaccessibles hors des méthodes de la classe (ou des fonctions ou classes amies).

Exemple :

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

class point { public : int abs, ord;
        point(int x=0, int y=0) : abs(x), ord(y) {}
        void affiche() {cout<<abs<<" "<<ord<<" ";}};

class pixel : protected point {
        public : string couleur;
        pixel(int x, int y, string c) : point(x,y), couleur(c) {}
        void affiche_pixel() {affiche(); cout<<couleur<<" ";}};

class pixel_clignotant : protected pixel {
        public : double frequence;
        pixel_clignotant(int x, int y, string c, double f)
                                        : pixel(x,y,c), frequence(f) {};
        pixel_clignotant(pixel pix, double f) : pixel(pix), frequence(f) {}
        void affiche_pix_c() {affiche_pixel(); cout<<frequence<<" ";}};

int main() {pixel A(5,68,"rouge");
        pixel_clignotant B(A, 5.5);
        A.affiche_pixel();
        cout<<endl;
        B.affiche_pix_c();
        cout<<endl;}
```

(nom du fichier : Chapitre4/prog02.cpp*)*

Exercice 1. *Quelles lignes du programme ci-dessous peut-on décommenter ?*

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

class point { public : int abs, ord;
        point(int x=0, int y=0) : abs(x), ord(y) {}};

class pixel : public point {
        public : string couleur;
        pixel(int x, int y, string c) : point(x,y), couleur(c) {}};

class pixel_clignotant : protected pixel {
        private : double frequence;
        public :
        pixel_clignotant(pixel pix, double f) : pixel(pix), frequence(f) {}
//      string get_col() {return couleur;}
        };

class pixel_cli_sonor : private pixel_clignotant {
        public : string note;
        pixel_cli_sonor(pixel_clignotant pic,  string n)
                                        : pixel_clignotant(pic), note(n) {}
//      double get_frequence() {return frequence;}
};

int main() {pixel A(5,68,"rouge");
//      cout<<A.abs<<endl;
        pixel_clignotant B(A, 5.5);
//      cout<<B.abs<<endl;
        pixel_cli_sonor C(B, "do");
//      cout<<C.frequence<<endl;
        }
```

(nom du fichier : Chapitre4/prog03.cpp*)*

1.2 Héritage multiple

La création de classe par le mécanisme d'héritage d'une classe préexistante se généralise : une nouvelle classe peut hériter de plusieurs classes préexistantes. La syntaxe est la suivante.

`class ma_classe_derivee : public : ma_classe_de_base_1, public : ma_classe_de_base_2`,... { déclaration } ;

Exemple :

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

class point { public : int abs, ord;
        point(int x=0, int y=0) : abs(x), ord(y) {}
        void afficher() {cout<<abs<<" "<<ord<<endl;} };

class couleur {
        public : string coul;
        couleur(string c) : coul(c) {}
        void afficher() {cout<<coul<<endl;} };

class pixel : public point, public couleur {public : bool visible;
  pixel(point p, couleur c, bool b) : point(p), couleur(c), visible(b) {}
  void afficher() {visible ? cout<<"visible " : cout<<"non visible ";} };

int main() {point p(2,3);
        p.afficher();
        couleur c("vert");
        c.afficher();
        pixel pix(p,c,true);
        pix.afficher();
        pix.point::afficher();
        pix.couleur::afficher();}
```

(nom du fichier : `Chapitre4/prog04.cpp`*)*

Ce type de notion est proche des mathématiques car de nombreuses objets ont une structure multiple : les permutations sont à la fois des fonctions qui agissent sur un ensemble, un groupe et peuvent avoir une théorie des représentations...

1.3 Opérateur global de résolution : :

Il arrive qu'un objet appartienne en même temps, par le biais de l'héritage, à plusieurs classes possédant des méthodes du même nom. Dans ce cas, pour savoir quelle fonction utiliser, on utilise l'opérateur global de résolution : : précédé du nom de la classe prioritaire de la méthode. Exemple :

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

class point { public : int abs, ord;
        point(int x=0, int y=0) : abs(x), ord(y) {}
        void afficher() {cout<<abs<<" "<<ord<<endl;} };

class pixel : public point {
        public : string couleur;
        pixel(point p, string c) : point(p), couleur(c) {}
        void afficher() {cout<<couleur<<endl;} };

int main(){point p(5,6);
        p.afficher();
        pixel pix(p,"jaune");
        pix.afficher();
        pix.point::afficher();}
```

(nom du fichier : `Chapitre4/prog05.cpp`*)*

1.4 Conversion standard vers une classe de base

Si une classe, appelée par exemple `pixel`, dérive d'une classe appelée par exemple `point`, alors tout objet de la classe `pixel` est un objet de la classe `point` auquel on a ajouté des caractéristiques et des possibilités. Par conséquent, selon le proverbe *Qui peut le plus peut le moins*, il est possible de convertir tout objet de la classe `pixel` en un objet de la classe `point`. La conversion se fait comme des `double` vers les `int` (partie entière) :

```
pixel pix;

point p=pix;
```

Exemple :

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

class point { public : int abs, ord;
        point(int x=0, int y=0) : abs(x), ord(y) {}
        void afficher() {cout<<abs<<" "<<ord<<endl;} };

class pixel : public point {
        public : string couleur;
        pixel(point p, string c) : point(p), couleur(c) {}
        void afficher() {cout<<couleur<<endl;} };

int main(){point p(5,6);
        pixel pix(p,"jaune");
        point q=pix;
        pix.afficher();
        q.afficher();}
```

(nom du fichier : `Chapitre4/prog06.cpp`*)*

Exercice 2 (Conversion standard d'une classe dérivée vers une classe de base)**.** *Donner la sortie du programme suivant.*

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

class B { public : B() {}
        void afficher() {cout<<"affichage de B"<<endl;} };

class D : public B { public : D() {}
        void afficher() {cout<<"affichage de D"<<endl;} };

int main() { D unD;
 B unB=unD;
 unD.afficher();
 unB.afficher(); }
```

(nom du fichier : `Chapitre4/prog07.cpp`*)*

2 Héritage et pointeurs : fonctions virtuelles

2.1 Type statique et type dynamique d'un pointeur ou d'une référence

Si une classe, appelée par exemple `pixel`, dérive d'une classe appellée par exemple `point`, alors tout objet de la classe `pixel` est un objet de la classe `point` auquel on a ajouté des caractéristiques et des possibilités. Par conséquent, tout pointeur vers un `pixel` est aussi l'adresse d'un `point` (sans conversion). Les pointeurs vers les `point` sont donc de plusieurs sortes : ceux dont le contenu n'est qu'un `point`, et ceux dont le contenu est un `pixel`. Cette subtilité est à la base des notions de *type statique* et *type dynamique*, elles mêmes à la base des notions de fonctions virtuelles, que nous verrons plus loin.

Exemple :

```
pixel unpixel;
point unpoint = unpixel;
```

```
point *ptrpoint1 = &unpixel;
point *ptrpoint2 = new pixel;
point &refpoint = unpixel;
```

Ici, on dit que le *type statique* de `*ptrpoint1`, `*ptrpoint2` et de `refpoint` est `point`, car c'est comme ça que `ptrpoint1`, `ptrpoint2` et `refpoint` ont été *déclarées*, mais que leur *type dynamique* est `pixel` car c'est le type des *valeurs effectives* de ces variables.

Exercice 3 (Type statique et type dynamique)**.** *Ici, on suppose* `B` *est une classe dont dérive une classe* `D` *(c'est à dire que* `D` hérite de `B`*).*

Dans les lignes suivantes, quels sont les types statiques et dynamiques des différents objets ?

```
D unD;
B unB = unD;
B *ptrB1 = &unD;
B *ptrB2 = new D;
B &refB = unD;
```

2.2 Priorité du type statique sur le type dynamique

Le type statique a priorité sur le type dynamique dans l'appel aux méthodes : si `ptr` a été déclaré comme un pointeur de `point` mais se trouve avoir pour contenu un `pixel`, pour toute méthode `f(...)` définie à la fois dans la classe `point` et dans la classe `pixel`,

```
p->f(...)
```

appellera la définition de `f(...)` donnée dans la classe `point`.

Exercice 4 (Priorité du type statique sur le type dynamique)**.** *Que va donner le programme suivant ? (l'essentiel de la réponse est contenue dans le titre de l'exercice...)*

NB : Il n'est pas évident de le prévoir, mais le programme ne compilera pas si l'on insère `pixel::` *devant la dernière occurrence de* `afficher()`*.*

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

class point {double abs, ord;
  public : point(double x=0, double y=0) : abs(x), ord(y) {}
  void afficher() {cout<<"point ("<<abs<<","<<ord<<")"<< endl;} };

class pixel : public point {string couleur;
  public :
  pixel(double x=0, double y=0, string c="") : point(x,y), couleur(c)  {}
  void afficher() {cout<<"pixel de couleur "<<couleur <<" en ";
                   point::afficher();} };

class point_clignotant : public point { double frequence;
  public :
  point_clignotant(double x=0, double y=0, double f=1) : point(x,y),
                                                  frequence(f)  {};
  void afficher() {cout<<"point clignotant de frequence "<<frequence <<" en ";
                       point::afficher(); } };

int main() {pixel *q=new pixel(3,3,"jaune");
        q->afficher();
        q->point::afficher();
        point *p=new pixel(2,2,"jaune");
        point *image[2];
        image[0]=new pixel(1,2,"rouge");
        image[1]=new point_clignotant(1,1,10);
        image[0]->afficher();
        image[1]->afficher();
        p->afficher();}
```

(nom du fichier : `Chapitre4/prog08.cpp`*)*

2.3 Fonctions virtuelles

Certaines fonctions d'une classe de base sont redéfinies dans ses classes dérivées et sont souvent appelées à travers des pointeurs ou des références d'objets des classes dérivées. La règle édictée au paragraphe précédent (*priorité du type statique sur le type dynamique*) est alors une contrainte qui oblige à faire souvent appel à l'opérateur global de résolution : :. Pour éviter cela, on place alors le mot clé `virtual` avant la déclaration de ces fonctions dans la classe de base : elles deviennent alors de *fonctions virtuelles* (appelées parfois *polymorphismes*). Pour de telles fonctions, **le type dynamique prend la priorité sur le type statique**.

Par exemple, on peut changer le programme précédent de façon à ce que les trois derniers affichages correspondent bien à des `pixel` et `point_clignotant` et non à des simples `point` : il suffit d'ajouter `virtual` devant la définition de `afficher` dans `point` :

```
virtual void afficher() {cout<<"point "<<abs<<""<<ord<<endl;};
```

En règle générale, les diverses redéfinitions d'une fonction `f(...)` peuvent être vues comme des versions de plus en plus spécialisées d'un traitement spécifié de manière générique dans la classe de base. Dans ces conditions, appeler une fonction virtuelle `f(...)` à travers un pointeur `ptr` sur un objet de la classe de base, c'est demander l'exécution de la version de `f(...)` la plus spécialisée possible.

Exemple (tiré du programme de gestion des stocks d'un magasin de vêtements) :

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

class produit { public : string nom;
        produit(string n) : nom(n) {}
        virtual void afficher() {cout<<nom<<endl;}};

class jean : public produit {public : int W,L;
        jean(string n, int w, int l) : produit(n), W(w), L(l) {}
        void afficher() {cout<<nom<<", "<<W<<" "<<L<<endl;} };

class chaussure : public produit {public : int taille;
        chaussure(string n, int t) : produit(n), taille(t) {}
        void afficher() {cout<<nom<<", "<<taille<<endl;} };

int main(){produit *stock[2];
        stock[0]=new jean("levi's 501", 33, 36);
        stock[1]=new chaussure("converse noire", 44);
        stock[0]->afficher();
        stock[1]->afficher();}
```

(nom du fichier : `Chapitre4/prog09.cpp`*)*

Contraintes : En concurrence avec le mécanisme de surcharge, la redéfinition des fonctions virtuelles subit des contraintes fortes, sur les types des paramètres et le type du résultat :

1. Tout d'abord, la signature (liste des types des arguments, sans le type du résultat) de la redéfinition doit être strictement la même que celle de la première définition, sinon la deuxième définition n'est pas une redéfinition mais un masquage pur et simple de la première.

2. Si une fonction virtuelle `f(...)` rend l'adresse d'un `T`, toute redéfinition de `f(...)` doit rendre l'adresse d'une sorte de `T` (i.e. d'un `T` ou d'une de ses classes dérivées) ; si `f(...)` rend une référence sur un `T`, toute redéfinition de `f(...)` doit rendre une référence sur une sorte de `T` (i.e. d'un `T` ou d'une de ses classes dérivées) ; enfin, si `f(...)` ne rend ni un pointeur ni une référence, toute redéfinition de `f(...)` doit rendre la même chose que `f(...)`.

3 Fonctions virtuelles pures et classes abstraites

Les fonctions virtuelles sont souvent introduites dans des classes placées à des niveaux si élevés de la hiérarchie d'héritage qu'on ne peut pas leur donner une implémentation utile ni même vraisemblable. Imaginons par exemple une classe `Figure` qui serait créée pour pouvoir regrouper, via l'héritage, dans un même type des pointeurs vers des objets de classes très différentes que seraient `triangle`, `cercle`, `ellipse`,...

On voudrait définir dans la classe `Figure` une fonction `afficher`, qui serait bien entendu virtuelle, car dépendant entièrement du type d'objet à dessiner (un `triangle` ne se dessinant pas comme un `cercle`, etc...).

L'idée est alors de définir dans la classe `Figure` une *fonction virtuelle pure*

```
virtual void afficher() = 0;
```

Cette démarche aura deux effets :

— la fonction `afficher` ne sera jamais appelée en tant que telle, mais **obligera** les programmeurs à mettre une version de cette fonction dans toute classe dérivée,

— elle fera de la classe où elle a été déclarée (la classe `Figure`) une *classe abstraite* : une classe qui n'est le type dynamique d'aucune adresse. Autrement dit, il peut exister des pointeurs vers des éléments de la classe `Figure`, mais pas d'objets de cette classe.

Exemple :

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

class point {public : double abs, ord;
        point(double x=0, double y=0) : abs(x), ord(y) {}
        friend std::ostream& operator<<(std::ostream &, const point &);};

std::ostream &operator<<(std::ostream &out, const point &p) {
    out <<" ("<<p.abs<<","<<p.ord<<") ";
    return out;};

class Figure { public :
        static int nombre_de_figures;
        Figure() {nombre_de_figures++;}
        virtual ~Figure() {nombre_de_figures--;}
        virtual void afficher() = 0;};

int Figure::nombre_de_figures = 0;

class segment : public Figure {public : point A,B;
        segment(point A, point B) : A(A), B(B) {};
        void afficher() {cout<<"Segment "<<A<<" "<<B<<endl;}};

class triangle : public Figure {public : point A,B,C;
        triangle(point A, point B, point C) : A(A), B(B), C(C) {};
        void afficher() {cout<<"Triangle "<<A<<" "<<B<<" "<<C<<endl;}};

int main() {point A(1,1), B(2,2), C(3,4);
        cout<<Figure::nombre_de_figures<<endl;
        Figure *ptr [2];
        cout<<Figure::nombre_de_figures<<endl;
        ptr[0]=new segment(A,B);
        cout<<Figure::nombre_de_figures<<endl;
        ptr[0]->afficher();
        ptr[1]=new triangle(A,B,C);
        cout<<Figure::nombre_de_figures<<endl;
        ptr[1]->afficher();
        delete ptr[0];
        delete ptr[1];
        cout<<Figure::nombre_de_figures<<endl;}
```

(nom du fichier : `Chapitre4/prog10.cpp`*)*

Remarque importante : il est fréquent que l'on utilise dans une classe un champ qui soit un pointeur vers une ojet d'une classe virtuelle pure. Il faudra faire très attention, comme l'indique le code suivant, à manier les constructeurs de cette classe avec précaution afin de ne jamais écrire un `new` sur une classe virtuelle pure mais seulement sur ses filles non virtuelles pures.

Exemple :

```
class A { public: virtual int methode(void) =0; };

class Aderiv1: public A { public: int methode(void) {return 1;}; };

class B {
                A * ptr_vers_A;
        public:
                //B(void): ptr_vers_A(new A) {};
                /* PROBLEME car allocation d'un objet
                 * d'une classe virtuelle pure */
};
```

(nom du fichier : `Chapitre4/prog11.cpp`*)*

4 Exercices

Exercice 5 (Conversion standard d'une classe dérivée vers une classe de base)**.** *Que va rendre le programme suivant ? A la ligne* `12`*, que se passe-t-il si l'on remplace* `Point::afficher();` *par* `afficher();` *?*

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

class Point {double abs, ord;
        public : Point(double x=0, double y=0) : abs(x), ord(y) {}
        void afficher() {cout<<"Point ("<<abs<<","<<ord<<")"<<endl;}};

class Pixel : public Point {string couleur;
        public : Pixel(double x=0, double y=0, string c="") : Point(x,y),
        couleur(c)  {};
        void afficher() {cout<<"Pixel defini comme suit :"<<endl;
        Point::afficher(); //ou afficher(); ?
        cout<<"Couleur : "<<couleur<<endl;}};

int main() {Pixel Pix(1,2,"rouge");
        Pix.afficher();
        Point P=Pix;
        P.afficher();}
```

(nom du fichier : `Chapitre4/prog12.cpp`*)*

Exercice 6 (Priorité du type statique sur le type dynamique, fonctions virtuelles)**.** *Donner la sortie du programme suivant. Comment modifier le programme pour que le deuxième affichage corresponde à celui de la classe* `D` *?*

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

class B { public : B() {};
        void afficher() {cout<<"affichage de B"<<endl;} };

class D : public B { public : D() {};
        void afficher() {cout<<"affichage de D"<<endl;} };

int main() { D unD;
 unD.B::afficher();
 B *pBD=new D;
 pBD->afficher();}
```

(nom du fichier : `Chapitre4/prog13.cpp`*)*

Exercice 7 (Fonctions virtuelles, priorité du type statique sur le type dynamique)**.** *Donner la sortie du programme suivant. Comment modifier le programme pour que le deuxième affichage corresponde à celui de la classe* `B` *?*

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

class B { public : B() {};
        virtual void afficher() {cout<<"affichage de B"<<endl;} };

class D : public B { public : D() {};
        void afficher() {cout<<"affichage de D"<<endl;} };

int main() { D unD;
 unD.B::afficher();
 B *pBD=new D;
 pBD->afficher(); }
```

(nom du fichier : `Chapitre4/prog14.cpp`*)*

Exercice 8 (Une fonction virtuelle pure ⇒ la classe est abstraite)**.** *Quelle ligne peut-on décommenter dans le programme suivant ? Comment modifier la déclaration de* `var_alea` *pour que l'on puisse décommenter les deux lignes ?*

```
class var_alea { public: var_alea()  {};
   virtual ~var_alea() {};
   virtual double operator()() = 0 ; };

int main() { //var_alea x;
        //var_alea * y;
}
```

(nom du fichier : `Chapitre4/prog15.cpp`*)*

Exercice 9 (Classes abstraites et fonctions virtuelles pures)**.** *a) Que va donner le programme suivant ?*

b) Peut-on décommenter la commande `Point unPoint;` *dans la fonction* `main` *? Peut-on le faire en décommantant la ligne* `Point() {};` *dans la définition de la classe* `Point` *?*

c) Peut-on décommenter la commande `ptr->afficher();` *dans la fonction* `main`*, quitte à décommenter la ligne* `Point() {};` *dans la définition de la classe* `Point` *?*

d) Le programme compilera-t-il si l'on remplace `q->afficher();` *par* `q->Point::afficher();` *à la deuxième ligne de* `main` *?*

e) Que doit-on changer dans la définition de la fonction `afficher` *de la classe* `Point` *pour que le remplacement de* `q->afficher();` *par* `q->Point::afficher();` *à la deuxième ligne de* `main` *n'empêche pas la compilation ?*

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

class Point {public : //Point() {}
                virtual void afficher() = 0;};

class Pixel : public Point {double abs, ord; string couleur;
        public : Pixel(double x=0, double y=0, string c="") : abs(x), ord(y),
                couleur(c)  {}
        void afficher()
 {cout<<"Pix. de coord. "<<abs<<" "<<ord<<" et coul. "<<couleur<<endl;}};

class Point_clignotant : public Point {double abs, ord; double frequence;
        public : Point_clignotant(double x=0, double y=0, double f=1) :
        abs(x), ord(y), frequence(f)  {}
                void afficher() {cout<<"Point cli. coord. "<<abs
                        <<", "<<ord<<" et freq. "<<frequence<<endl;}};

int main() {Point *ptr;
        //ptr->afficher();
        //Point unPoint;
        Pixel *q=new Pixel(3,3,"jaune");
        q->afficher();

        Point *p=new Pixel(2,2,"jaune");
        Point *image[2];
        image[0]=new Pixel(1,2,"rouge");
        image[1]=new Point_clignotant(1,1,10);
        image[0]->afficher();
        image[1]->afficher();
        p->afficher();}
```

(nom du fichier : `Chapitre4/prog16.cpp`*)*

Exercice 10 (Classes abstraites et fonctions virtuelles pures)**.** *Écrire deux classes dérivées*

`class uniform : public var_alea` *et* `class expo : public var_alea`

(correspondant aux variables uniformes et exponentielles, pour lesquelles on choisi, dans les membres privés, respectivement les bornes du support et le paramètre) de la classe suivante. Les tester et tester, via Scilab, les histogrammes des échantillons obtenus (voir la feuille 1 pour les instructions Scilab idoines).

```
class var_alea { public : var_alea()  {};
   virtual ~var_alea() {};
   virtual double operator()() = 0;};
```

(nom du fichier : `Chapitre4/prog17.cpp`*)*

Exercice 11 (Classes abstraites et fonctions virtuelles pures)**.** *Écrire une classe* `Point` *(élémentaire) ainsi que deux classes dérivées*

`class Triangle : public Figure` *et* `class Cercle : public Figure`

de la classe suivante. Les tester.

```
class Figure { public : static int nombre_de_Figures;
        Figure() {nombre_de_Figures++;
                        std::cout<<"on cree une figure !"<<std::endl;};
        virtual ~Figure() {nombre_de_Figures--;};
        virtual void afficher() =0;
        virtual void deplacer(double x, double y) = 0;
        virtual double aire() = 0;
        virtual double perimetre() = 0; };
```

(nom du fichier : `Chapitre4/prog18.cpp`*)*

On pourra utiliser la *formule de Héron*, quit dit que la surface S d'un triangle dont les côtés ont pour longueurs a, b, c vaut

$$S = \sqrt{q(q-a)(q-b)(q-c)},$$

où $q := \frac{a+b+c}{2}$ est le demi-périmètre.

Exercice 12 (exemple algébrique)**.** *Écrire une classe abstraite* `Groupe` *avec une loi de composition interne et une inversion. Écrire une classe dérivée* `Permutation` *qui code les permutations et leurs compositions.*