Surdéfinition d'opérateurs

Mécanisme de la surdéfinition d'opérateurs

Surdéfinition d'opérateur en général

L'opérateur =

Forme canonique

Surdéfinition d'opérateur

- Il est possible de redéfinir les opérateurs (+, -, …)
  - Consiste simplement en l'écriture de nouvelles fonctions surdéfinies
  - Possible uniquement lorsque l'opérateur porte sur au moins un objet
- Syntaxe: mot-clé *operator* suivi de l'opérateur concerné
  - Exemple: *operator+*
- Peut être effectué à l'aide
  - D'une fonction amie de la classe concernée
  - D'une fonction membre de la classe concernée
    - Le premier opérande sera l'objet ayant appelé la fonction membre

Exemple fonction amie

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        ~Point();
        friend Point operator+(Point, Point);
};
Point operator+(Point p1, Point p2){
    Point p(p1.x+p2.x, p1.y+p2.y);
    return p;
}
```

```
main(){
    Point a(4, 5), b(5, 6);
    Point c = a + b;
}
```

Ici, l'opérateur + est surdéfini par une fonction amie de la classe *Point*. Les opérandes sont les deux attributs de type *Point*. Cette fonction peut être appelée par l'opérateur + appelé avec deux opérandes de type *Point*.

Exemple fonction membre

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        ~Point();
        Point operator+(Point);
};
Point Point::operator+(Point p2){
    Point p(x+p2.x, y+p2.y);
    return p;
}
```

```
main(){
    Point a(4, 5), b(5, 6);
    Point c = a + b;
}
```

Ici, l'opérateur + est surdéfini par une fonction membre de la classe *Point*. Le premier opérande est l'objet de type *Point* qui va appeler la fonction, et le second opérande est l'argument de type *Point*. L'appel de cette fonction par l'opérateur + est similaire à la fonction amie.

Opérateur et référence

- Il est possible de faire appel à la transmission d'argument par référence
  - En particulier dans le cas d'objet de grande taille

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        ~Point();
        friend Point operator+(const Point &, const Point &);
};
```

```
Point operator+(const Point &p1, const Point &p2){
    Point p(p1.x+p2.x, p1.y+p2.y);
    return p;
}
```

L'avantage ici est que des références sur les opérandes de type *Point* seront utilisées, et aucune recopie d'un argument ne sera effectuée. L'inconvénient est que les arguments peuvent être modifiée par l'opérateur +. Le mot clé *const* peut être utilisé pour éviter ce problème.

Surdéfinition d'opérateurs

- Le symbole suivant *operator* doit être forcement un opérateur existant
  - Pas possible de créer de nouveaux symboles
- Certains opérateurs ne peuvent pas être redéfinis
  - Opérateurs point « . », de résolution de porté « :: » et conditionnel « ?: »
- Il faut conserver la pluralité (unaire, binaire) de l'opérateur initial
- Les priorités relatives des opérateurs initiaux sont conservées

Opérateurs surdéfinissables

Plus prioritaire

Moins prioritaire

Catégorie	Opérateurs	Priorité
Référence	*() [] ->*	→
Unaire	*+ - ++ -- ! (cast) new new[] delete delete[]*	←
Arithmétique	** / %*	→
Arithmétique	*+ -*	→
Relationnels	*< <= > >=*	→
Relationnels	*== !=*	→
Logique	*&&*	→
Logique	*||*	→
Affectation	*= += -= *= /= %=*	←

Règles

TD 4
Exercice 4

- L'opérateur surdéfini doit comporter un argument de type classe
- Certains opérateurs doivent être définis comme membres d'une classe
  - Il s'agit de *[]*, *()*, -> ainsi que *new* et *delete*

- Certains opérateurs possèdent une signification par défaut
  - *->*, *new*, *new[]*, *delete*, *delete[]* et =
- Il est possible d'attribuer à chaque opérateur la signification voulue

  - Opérateur d'affectation élargie pas défini par = ou l'opérateur associé
    - Surdéfinir = ou + ne change pas la signification de +=

Cas de l'opérateur d'incrémentation

- Peut définir l'opérateur ++ (ou --) en notation préfixe, et en postfixe
  - Ajouter un argument fictif supplémentaire de type *int* à la version postfixe
    - Aucune valeur réellement transmise

- Surdéfinir une notation ne surdéfinit pas l'autre notation

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        Point operator++();
        Point operator++(int);
};
```

```
Point Point::operator++(){
    x++; y++;
    return *this;
}
Point Point::operator++(int fictif){
    Point p_old = *this;
    x++; y++;
    return p_old;
}
```

L'opérateur =

- Recopie par défaut les valeurs des champs du 2^{nd} opérande dans le 1^{er}
  - Insatisfaisant si les objets comportent des pointeurs alloués dynamiquement
- Situation similaire au constructeur de recopie mais pas identique

  - Il peut y avoir affectation d'un objet à lui-même
  - Avant affectation, il existe deux objets (avec leurs partie dynamiques)
- Pour traiter l'affectation multiple, l'opérateur = doit renvoyer une valeur
  - Situation du type *a = b = c;*

Algorithme proposé

- Si les deux opérandes sont différents
  - Libérer les emplacements dynamiques alloués par le premier opérande
  - Création dynamique de nouveaux emplacements pour le premier opérateur
    - Pour recopier les valeurs du second opérande
- Si les deux opérandes correspondent au même objet
  - Si la transmission du second opérande est par référence
    - Il ne faut surtout pas libérer les emplacements dynamiques de l'objet
  - Il vaut mieux ne rien faire

```
class Vect{
    private:
        int taille, *vec;
    public:
        Vect(int);
        ~Vect();
        Vect(Vect &);
        Vect operator=(const Vect &);
};
Vect::Vect(int t){
    taille = t;
    vec = new int[taille];
}
Vect::~Vect(){
    delete [] vec;
}
```

```
Vect::Vect(Vect &v){
    taille = v.taille;
    vec = new int[taille];
    for(int i = 0; i < taille; i++)
        vec[i] = v.vec[i];
}
Vect Vect::operator=(const Vect &v){
    if(this != &v){
        delete [] vec;
        taille = v.taille;
        vec = new int[taille];
        for(int i = 0; i < taille; i++)
            vec[i] = v.vec[i];
    }
    return *this;
}
```

Dans la fonction membre *operator=*, *this* est un poiteur (contenant une adresse) sur le premier opérande et *&v* est l'adresse du second opérande. Le premier test consiste à regarder si les deux opérandes sont différant, c'est-à-dire si ils n'ont pas la même adresse. Si c'est bien le cas, les champs dynamiques du second opérande sont libérés, puis réservé de nouveau avec la même taille que celle du premier

Référence et valeur de retour

- Une fonction peut transmettre sa valeur de retour par référence
  - Ne doit pas faire référence à une variable locale
  - La valeur de retour peut être utilisée comme une lvalue

```
class Vect{
    private:
        int taille, *vec;
    public:
        Vect(int);
        ~Vect();
        Vect(Vect &);
        Vect & operator=(const Vect &);
};
```

```
Vect & Vect::operator=(const Vect &v){
    if(this!=&v){
        delete [] vec;
        taille = v.taille;
        vec = new int[taille];
        for(int i = 0; i<taille; i++)
            vec[i] = v.vec[i];
    }
    return *this;
}
```

Exemple

```
class Vect{
    private:
        int taille, *vec;
    public:
        Vect(int);
        ~Vect();
        Vect(Vect &);
        Vect & operator=(Vect &);
        int & operator [ ](int);
};
```

```
int & Vect::operator [ ](int i){
    return vec[i];
}
main(){
    Vect v(5);
    v[3] = 5;
}
```

Ici, *v[3]* fait appel à la fonction membre *operator[]*, qui va renvoyer une référence vers la 4^{eme} case du vecteur. Il est donc possible d'affecter une valeur au retour de cette fonction car il s'agit d'une référence et non une valeur. Notez que ça ne serait pas possible si l'opérateur *[]* ne renvoyait pas une référence

Forme canonique

TD 4bis

- Lorsqu'une classe dispose de pointeurs sur des parties dynamiques
  - Pour que la recopie d'objet fonctionne convenablement il faut redéfinir
    - Constructeur, destructeur, constructeur de recopie et opérateur d'affectation

```
class T{
        …
        public:
                T(…);                          //constructeur
                T(const T &);                  //constructeur de recopie
                ~T();                          //destructeur
                T & operator=(const T &);      //opérateur d'affectation
                …
};
```

Héritage

Notion d'héritage

Redéfinition de fonction

Contrôle d'accès

Fonctions amies

Polymorphisme et fonctions virtuelles

Classe abstraite

Héritage

- L'héritage est l'un des fondements de la programmation orienté objet
- Possibilité de réutilisation de composants logiciels
  - Autorise à définir une classe dérivée à partir d'une classe de base
    - La classe dérivée hérite des potentialités tout en y ajoutant de nouvelles
  - Plusieurs classes pourront hériter d'une même classe de base
  - Une classe dérivée peut devenir à son tour classe de base pour une autre classe

Notion d'héritage

```
class Point{
    private:
        int x,y;
    public:
        void initialise(int, int);
        bool coincide(Point);
};
```

```
class Point3D : public Point{
    private:
        int z;
    public:
        void initilialise3D(int, int, int);
        bool coincide(Point3D);
};
Point3D p1, p2;
p1.initialise3D(4, 5, 6);
p2.initialise(4, 5);
```

Un objet de la classe *Point3D* est un *Point* avec un attribut supplémentaire z, et avec deux fonctions membres supplémentaires. *p1* fait appel à une des fonctions de la classe *Point3D* alors que *p2* fait appel à une fonction de la classe *Point*.

- La déclaration de *Point3D* spécifie que la classe dérive de *Point*
  - Les membres *public* de la classe de base sont aussi *public* dans la classe dérivée
  - Chaque objet de la classe dérivée peut appeler les fonctions de celle de base

Membres privés et héritage

- La classe dérivée n'a pas accès aux membres privés de sa classe de base
  - Elle n'a accès qu'aux membres publiques de sa classe de base

```
void Point3D::initialise3D(int a, int b, int c){
    initialise(a, b);
    z = c;
}
```

Ici, l'initialisation des attributs x et y est faite par l'intermédiaire de la fonction membre publique *initialise* qui est héritée de la classe *Point* car les membres x et y ne sont pas accessibles dans la classe *Point3D*.

Redéfinition de fonction

- Il est possible de redéfinir des fonctions de la classe de base
  - L'opérateur :: permet de forcer l'appel à la fonction de la classe de base

```
bool Point3D::coincide(Point3D p){
    return (z == p.z) && Point::coincide(p);
}
Point3D p1, p2;
p1.initialise3D(4, 5, 6);
p2.initialise(4, 5);
p1.coincide(p2);
p2.Point::coincide(p1);
```

Ici, l'instruction *p1.coincide(p2)* fait appel à la fonction membre *coincide* redéfinie dans la classe *Point3D*, alors que l'instruction *p2.Point::coincide(p1)* fait appel à la fonction membre de la classe *Point* (un *Point3D* étant également un *Point*).

Redéfinition de fonction

- Lorsqu'une fonction membre est défini dans une classe
  - Elle masque toutes les fonctions membres de même nom de la classe de base
    - *p1.coincide(p3)* aurait été rejeté si *p3* était de la classe *Point*
  - Peut imposer que le recherche de fonction se fasse dans plusieurs classes
    - Introduit *using Point::coincide;* dans la définition (niveau *public*) de la classe *point3D*

Héritage et constructeur

- La classe de base peut posséder un ou plusieurs constructeurs
  - Un constructeur de la classe dérivée doit faire appel à un de ces constructeurs
- L'appel est automatique pour un constructeur sans paramètre
  - Sinon les paramètres doivent être précisés dans la définition du constructeur

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
};
```

```
class Point3D : public Point{
    private:
        int z;
    public:
        Point3D(int, int, int);
};
Point3D::Point3D(int a, int b, int c): Point(a, b){
    z = c;
}
```

Héritage et destructeur

- Le destructeur de la classe de base ne peut pas avoir de paramètre
  - Il est appelé automatiquement après le destructeur de la classe dérivée
    - Le destructeur de la classe dérivée peut être omis

```
class Vect{
        private:
                int taille, *vec;
        public:
                Vect(int);
                ~Vect();
};
Vect::~Vect(){
        delete [] vec;
}
```

```
class Vect2 : public Vect{
        private:
                int taille2, *vec2;
        public:
                Vect2(int, int)
                ~Vect2();
};
Vect2::Vect2(int t1, int t2): Vect(t1){
        vec2 = new int[taille2 = t2];
}
Vect2::~Vect2(){
        delete [] vec2;
}
```

Membres *protected*

- Le statut *protected* est à mi-chemin entre *public* et *private*
  - Membres inaccessibles aux utilisateurs mais accessibles aux classes dérivées
  - Mot-clé qui s'emploie comme *public* et *private*

```
class Point{
    protected:
        int x,y;
};
class Point3D : public Point{
    private:
        int z;
    public:
        void initialise(int, int, int);
};
```

```
void Point3D::initialise(int a, int b, int c){
    x = a;
    y = b;
    z = c;
}
```

Ici, les attributs x et y ne sont pas accessibles en dehors de la classe *Point* (comme des attributs privés), mais ils sont accessibles dans la classe *Point3D* qui descend de la classe *Point*. C'est pour cette raison que la fonction membre *initialise* de *Point3D* peut les utiliser

Dérivation privée

- La dérivation publique conserve le statut des membres hérités
  - Les membres *public* ou *protected* conservent leurs statuts
- La dérivation privée interdit l'accès aux membres de la classe de base
  - Les membres *public* ou *protected* hérités deviennent *private*
    - Ces membres sont néanmoins accessibles au sein de la classe dérivée

```
class Point{
    protected:
        int x,y;
    public:
        void initialise(int, int);
        bool coincide(Point);
};
```

```
class Point3D : private Point{
    private:
        int z;
    public:
        void initilialise(int, int, int);
        bool coincide(Point3D);
};
```

Ici, les attributs x et y et les fonctions membres *initialise* et *coincide* prennent le statut *private* au sein de la classe *Point3D*. Ils sont accessibles dans la classe *Point3D*, mais ne seront pas accessibles aux classes qui héritent de la classe *Point3D*.

Dérivation protégée

- Il existe une possibilité intermédiaire qui est la dérivation protégée
  - Les membres *public* et *protected* de la classe de base deviennent *protected*

```
class Point{
    public:
        int x,y;
    public:
        void initialise(int, int);
        bool coincide(Point);
};
```

```
class Point3D : protected Point{
    private:
        int z;
    public:
        void initilialise(int, int, int);
        bool coincide(Point3D);
};
```

Ici, les attributs et les fonctions membres hérités ont le statut *protected* dans la classe *Point3D*. Ils ne pas sont accessibles hors de la classe *Point3D*, mais ils le seront dans la classe *Point3D* ainsi que dans les classes qui héritent de la classe *Point3D*.

Exceptions

- Il est possible dans une dérivation privée et protégée de conserver le statut *public* ou *protected* d'un membre de la classe de base
  - Il faut utiliser le mot clé *using* au sein de la classe dérivée

```
class Point{
    private:
        int x,y;
    public:
        void initialise(int, int);
        bool coincide(Point);
};
```

```
class Point3D : private Point{
    private:
        int z;
    public:
        using Point::initialise;
        void initilialise(int, int, int);
        bool coincide(Point3D);
};
```

Ici, les attributs et les fonctions membres hérités seront privés, excepté pour la fonction initialise qui sera publique.

Héritage et fonctions amies

- Les fonctions amies ont les mêmes autorisations que les fonctions membres
- Les déclarations d'amitiés ne s'héritent pas
  - *initialise* ne sera pas automatiquement amie des descendants de *Point3D*

```
class Point{
    protected:
        int x,y;
};
class Point3D : public Point{
    private:
        int z;
    public:
        friend void initialise(Point3D, int, int, int);
};
```

```
void initialise(Point3D p, int a, int b, int c){
    p.x = a;
    p.y = b;
    p.z = c;
}
```

TD 5

Classes de base et dérivées

- En programmation orienté objet, on considère qu'un objet de la classe dérivée peut « remplacer » un objet de la classe de base
  - Repose sur l'existence de conversion implicite
    - D'un objet de la classe dérivée dans un objet de la classe de base
    - D'un pointeur sur la classe dérivée en un pointeur sur la classe de base

Conversion d'objets

```
class Point{
    private:
        int x,y;
    public:
        void initialise(int, int);
        bool coincide(Point);
};
```

```
class Point3D : public Point{
    private:
        int z;
    public:
        bool coincide(Point3D);
};
Point3D p1;
Point p2;
p2 = p1;
```

- L'affectation ici est légale

  - Une conversion de *p1* en *Point* est effectuée avant l'affectation
    - Ne conserve de *p1* que ce qui est de classe *Point*

Conversion de pointeur


```
class Point{
    private:
        int x,y;
    public:
        void initialise(int, int);
        bool coincide(Point);
};
```

```
class Point3D : public Point{
    private:
        int z;
    public:
        bool coincide(Point3D);
};
Point3D *p1 = new Point3D;
Point *p2 = p1;
p2->coincide(*p2);
```

- L'affectation ici est légale
  - Une conversion de *p1* en *Point* * est effectuée avant l'affectation
- Le choix de la fonction membre dépend du type du pointeur
  - Ici, c'est la fonction de *Point* qui est choisie bien que *p2* pointe sur un *Point3D*

Héritage et constructeur de recopie

- Si la classe dérivée ne définit pas de constructeur de recopie
  - Le constructeur de recopie par défaut est appelé lors de la création d'un objet
    - Si la classe de base possède un constructeur de recopie, il est appelé
- Si la classe dérivée définit un constructeur de recopie
  - Le constructeur de recopie de la classe de base n'est pas appelé par défaut
    - Il est possible de l'appeler de manière similaire à un constructeur classique

Exemple

```
class Vect{
    private:
        int taille, *vec;
    public:
        Vect(int);
        ~Vect();
        Vect(const Vect &);
};
Vect::Vect(const Vect &v){
    vec = new int[taille = v.taille];
    for(int i = 0; i < taille; i++)
        vec[i] = v.vec[i];
}
```

```
class Vect2 : public Vect{
    private:
        int taille2, *vec2;
    public:
        Vect2(int, int)
        ~Vect2();
        Vect2(const Vect2 &)
};
Vect2::Vect2(const Vect2 &v): Vect(v){
    vec2 = new int[taille2 = v.taille2];
    for(int i = 0; i < taille2; i++)
        vec2[i] = v.vec2[i];
}
```

Ici, le constructeur de recopie de la classe *Vect2* est redéfini. Ce constructeur doit faire appel au constructeur de recopie de la classe *Vect* pour initialiser les champs *taille* et *vec* qui sont privés. Il prend comme paramètre *v* qui sera converti en référence sur un *Vect*.

Héritage et opérateur d'affectation

- Si la classe dérivée ne surdéfinit pas l'opérateur =
  - L'opérateur d'affectation par défaut est appelé
    - Si la classe de base a surdéfini l' opérateur =, alors l'opérateur surdéfini est appelé
- Si la classe dérivée surdéfinit l'opérateur =
  - L'opérateur d'affectation de la classe de base ne peut pas être appelé
    - On peut utiliser les possibilités de conversion de pointeurs

```
class Vect{
    private:
        int taille, *vec;
    public:
        Vect(int);
        ~Vect();
        Vect & operator=(const Vect &); }
};
class Vect2 : public Vect{
    private:
        int taille2, *vec2;
    public:
        Vect2(int, int);
        ~Vect2();
        Vect2 & operator=(const Vect2 &);
};
```

```
Vect & Vect::operator=(const Vect &v){
    if(this != &v){
        delete [] vec;
        vec = new int[taille = v.taille];
        for(int i = 0; i < taille; i++)
            vec[i] = v.vec[i];
    }
    return *this;
}
Vect2 & Vect2::operator=(const Vect2 &v){
    if(this != &v){
        Vect *p1 = this, *p2 = &v;
        *p1 = *p2;
        delete [] vec2;
        vec2 = new int[taille2 = v.taille2];
        for(int i = 0; i < taille2; i++)
            vec2[i] = v.vec2[i];
    }
    return *this;
}
```

L'opérateur = redéfini dans la classe *Vect2* fait appel à l'opérateur = de la classe *Vect* par l'intermédiaire de l'instruction **p1 = *p2*. *p1* étant un pointeur sur un *Vect*, c'est la fonction de la classe *Vect* qui est appelée.

Polymorphisme

- Un pointeur peut recevoir l'adresse d'un objet d'une classe dérivée
- L'appel d'une méthode dépend de la classe du pointeur
  - Type d'un objet déterminé au moment de la compilation
- Pour faire appel à la méthode correspondant au type de l'objet pointé
  - Il faut qu'il soit pris en compte au moment de l'exécution
    - On parle alors de polymorphisme

Fonction virtuelle

- Le mécanisme de fonction virtuelle va permettre le polymorphisme
  - Une fonction membre peut être déclarée *virtual* lors de sa déclaration
    - Indique que le choix de la fonction appelée doit se faire à l'exécution

```
class Point{
    private:
        int x,y;
    public:
        virtual bool coincide(Point);
};
```

```
class Point3D : public Point{
    private:
        int z;
    public:
        bool coincide(Point3D);
};
Point3D *p1 = new Point3D;
Point *p2 = p1;
p2->coincide(*p2);
```

La fonction *coincide est* définie dans la classe *Point* et redéfinie dans la classe *Point3D*. Cette fonction étant *virtual*, l'appel à partir du pointeur *p2*, qui pointe sur un objet de la classe *Point3D*, entraine l'appel de la fonction de la classe *Point3D*

Autre situation de polymorphisme

```
class Point{
    private:
        int x,y;
    public:
        virtual void identifie();
        void affiche();
};
class Point3D : public Point{
    private:
        int z;
    public:
        void identifie();
};
```

```
void Point::identifie(){
    cout << " point: " << x << " , " << y << '\n';
}
void Point::affiche(){
    cout << " je suis un ";
    identifie();
}
void Point3D::identifie(){
    cout << " point3D: " << x << " , ";
    cout << y << " , " << z << '\n';
}
Point3D p;
p.affiche();
```

La fonction *affiche* est appelée à partir d'un *Point3D*. Elle fait appel à la fonction *identifie* à partir d'un pointeur sur un *Point* (*this*) qui pointe sur un *Point3D*. La fonction *identifie* étant vitual, c'est celle de la classe Point3D qui est appelée.

Propriétés fonctions virtuelles

- Si une fonction est déclarée virtuelle dans une classe
  - Elle le sera également dans toute classe descendante de cette classe
  - Il n'est pas indispensable de la redéfinir dans toutes les classes descendantes
    - La redéfinition d'une fonction virtuelle doit respecter le type de la valeur de retour
- Seule une fonction membre peut être virtuelle
- Un constructeur ne peut pas être virtuel
- Un destructeur peut être virtuel
  - Conseillé d'avoir un destructeur virtuel en cas de polymorphisme

Exemple

```
class Vect{
    private:
        int taille, *vec;
    public:
        Vect(int);
        virtual ~Vect();
};
class Vect2 : public Vect{
    private:
        int taille2, *vec2;
    public:
        Vect2(int, int);
        ~Vect2();
};
```

```
Vect::~Vect(){
    delete [] vec;
}
Vect2::~Vect2(){
    delete [] vec2;
}
main(){
    Vect v* = new Vect2(10,10);
    delete v;
}
```

v est un pointeur sur un *Vect*, qui pointe sur un *Vect2*. Le mot clé *delete* fait appel au destructeur. Le destructeur choisit ici est celui de la classe *Vect2* car le destructeur est *virtual* et l'objet pointé est de la classe *Vect2*. Si le destructeur n'avait pas été *virtual* alors c'est celui de la classe *Vect* qui aurait été appelé.

Classe abstraite

- Classe abstraite: classe pas destinée à instancier des objets
  - Simplement à donner naissance à d'autres classes par héritage
- Les fonctions virtuelles pures permettent de définir des classes abstraites
  - Fonction virtuelle dont la définition est nulle (0)
  - Une classe comportant un fonction virtuelle pure est abstraite
  - Une classe dérivée d'une classe abstraite doit redéfinir toutes les fonctions virtuelles pures pour ne pas être abstraite

```
class Point{
    public:
        vitual void affiche() = 0;
};
class Point2D : public Point{
    private:
        int x, y;
    public:
        void affiche();
};
class Point3D : public Point{
    private:
        int x, y, z;
    public:
        void affiche();
};
```

```
class Vect{
    private:
        Point* vec[10];
    public:
        void affiche_all();
};
void Point2D::affiche(){
    cout << x << " , " << y << '\n';
}
void Point3D::affiche(){
    cout << x << " , " << y << " , " << z << '\n';
}
void Vect::affiche_all(){
    for(int i = 0; i < 10; i++)
        vec[i]->affiche();
}
```

La classe *Point* est abstraite car *affiche* est une fonction virtuelle pure. Les classes *Point2D* et *Point3D*, qui héritent de *Point*, ne sont pas abstraites car elles redéfinissent la fonction *affiche*.

Identification de type

TD 6
Exercice 1

- Permet de connaitre le type d'un objet désigné par pointeur ou référence
  - La fonction *typeid* fournit un objet de classe *type_info*
    - Cette classe contient une fonction membre *name()* qui renvoie le nom de la classe
      - Ce nom peut dépendre de l'implémentation
    - Les opérateurs == et *!*= permettent de comparer des objets de cette classe

```
Point *p1 = new Point2D, *p2 = new Point3D;
cout << typeid(p1).name() << " == " << typeid(p2).name();
cout << " = " << typeid(p1) == typeid(p2) << '\n';
```

Patrons de fonctions

Paramètres de type et expressions

Surdéfinition et spécialisation de patron

Patrons de fonctions

- Permettent d'écrire une fonction sans préciser le type de ses paramètres
  - Le compilateur adaptera la définition à partir du type de ses paramètres effectifs

```
template <class T> T min (T a, T b){
    return a < b ? a : b;
}
```

- La mot-clé *template* indique que l'on a affaire à un patron
  - Le paramètre de ce patron est *T* qui est introduit par le mot-clé *class*
    - *T* est un identifiant (comme le serait le nom d'une variable)

Utilisation d'un patron

- Pour utiliser un patron, il suffit d'utiliser la fonction de manière « appropriée »
  - Le compilateur « fabriquera » automatiquement la fonction correspondante
    - Les arguments peuvent être de n'importe quel type (pointeurs, structures, objets, …)
    - Attention: il faut que les opérations utilisées soient définies pour le type de l'argument
  - Attention: la correspondance des types doit être absolue
    - Ici le types des deux arguments doivent être les mêmes

```
int c = 1, d = 2;
min(c, d);
float e = 1.5, f = 2.5;
min(e, f);
```

Lorsque le compilateur rencontre l'appel de fonction *min(c,d)*, il crée une fonction *min* avec comme paramètres deux entiers. L'instruction *min(c,e)* serait incorrecte car c et e ne sont pas du même type. De même, il serait impossible d'appeler la fonction sur deux objets (de même type) pour lesquels l'opérateur < (utilisé dans le patron de fonction *min*) n'a pas été redéfini.

Contraintes d'utilisation

- Le patron doit être défini dans le fichier avant d'être appelé
- La définition d'un patron de fonction correspond en fait à une déclaration
  - Elle est utilisée par le compilateur pour fabriquer les fonctions requises
  - En pratique, il faut la placer dans un fichier d'en-tête d'extension .h

Paramètres de type

- Un patron peut comporter un ou plusieurs paramètres de type
  - Chacun devra être précédé du mot-clé *class*
  - Ces paramètres peuvent apparaitre dans l'en-tête ou dans les instructions
  - Attention: chaque paramètre doit apparaitre au moins un fois dans l'en-tête

```
template <class T, class U> T min(T a, U b){
    return a < b ? a : b;
}
int c = 4; float d = 3.5;
min(c, d);
```

L'appel *min(c,d)* est autorisé ici, malgré que c et *d* ne soit pas du même type, car les deux paramètres du patron *min* ont des types différents (*T* et *U*). Cet appel va créer une fonction dont le premier paramètre est un *int*, et dont le second paramètre est un *float*.

Identification des types

- C++ autorise la spécification d'un ou plusieurs types au moment de l'appel

```
int c = 4; float d = 3.5
min<int, float>(c, d);
min<float>(d, c);
```

Ici, on indique au compilateur dans l'instruction *min<int, float>(c, d)* que le premier argument sera de type *int* et que le second sera de type *float*.

- Le mode de transmission (valeur ou référence) ne joue aucun rôle
  - Ne peut pas être détecté lors de l'appel

Paramètres d'expression

TD 6
Exercice 2

- Les patrons de fonctions peuvent contenir des paramètres « ordinaires »

```
template <class T> T min_tab(T *tab, int taille){
    T minimum = tab[0];
    for(int i = 1; i < taille; i++)
        minimum = (minimum < tab[i] ? minimum : tab[i]);
    return minimum;
}
```

Le premier paramètre de ce patron est un pointeur vers un type paramétré *T* (qui est en fait un tableau d'éléments de ce type). Le second est un entier qui correspondra à la taille du tableau.

- On peut imposer qu'un paramètre de type soit un pointeur

Surdéfinition de patron

- Il est possible de définir plusieurs patrons portant le même nom
  - Ces patrons doivent avoir des arguments différents
  - Conduit à définir plusieurs « familles » de fonctions

```
template <class T> T min(T a, T b, T c){
    return min(min(a, b), c);
}
int d = 4, e = 3, f = 2;
min(d, e, f);
```

Ici, le patron *min* a trois paramètres du même type. Il fait appel au patron *min* avec deux paramètres dans sa définition.

- La définition d'un patron peut faire intervenir des fonctions partons

Spécialisation

- Peut spécifier la définition d'un patron pour certains types d'arguments

```
template <class T> T min (T a, T b){
        return a < b ? a : b;
}
char* min(char *a, char *b){
        return strcmp(a, b) < 0 ? a : b;
}
```

Ici, la définition du patron de classe *min* est spécialisée dans le cas où les arguments sont des pointeurs sur *char*. Si le patron est appelé avec ce type de paramètres, c'est cette seconde définition qui s'appliquera.

Spécialisation partielle

- Définir des familles de fonctions, certaines étant plus générales que d'autres

```
template <class T> T min (T a, T b){
        return a < b ? a : b;
}
template <class T>  T* min(T *a, T *b){
        return *a < *b ? a : b;
}
```

Ici, la définition du patron de classe *min* est spécialisée dans le cas où les arguments sont des pointeurs (sur n'importe que type). Si le patron est appelé avec un pointeur, c'est cette seconde définition qui s'appliquera.

Algorithme d'instanciation

- Si une ou plusieurs fonctions « ordinaires » ont une correspondance exacte
  - Si une seule elle est choisie
  - Sinon il y a ambiguïté (erreur de compilation)
- Sinon, tous les patrons ayant le nom voulu sont examinés
  - Si il y a correspondance exacte des paramètres de type pour un patron
    - La fonction correspondante est implémentée
  - Si il y a correspondance exacte des paramètres de type pour plusieurs patrons
    - Si un patron est plus spécialisé que les autres, alors il est implémenté
    - Sinon il y a ambiguïté (erreur de compilation)
  - Si pas de correspondance exacte des paramètres de type
    - Les fonctions « ordinaires » sont reconsidérées et traitées comme des fonctions surdéfinies

Patrons de classes

Paramètres de type et d'expression

Spécialisation

Patrons de classes

- C++ permet de définir des patrons de classes
  - Le compilateur adaptera la définition de la classe aux différents types

```
template <class T> class Point{
    private:
        T x, y;
    public:
        Point(T = 0, T = 0);
        Point<T> operator+ (Point<T>);
};
```

Ici, le patron de classe *Point* permet d'instancier des points dont le type des coordonnées (x et y) n'est pas défini à l'avance. Remarquez également que le type paramétré apparait dans le constructeur et la fonction membre

- La mot-clé *template* indique que l'on a affaire à un patron
  - Le paramètre de ce patron est *T* qui est introduit par le mot-clé *class*

Définition de fonctions membres

- Dans la définition des fonctions membres il faut fournir
  - La liste des paramètres de types (précédée du mot-clé *template*)
  - Le nom du patron concerné (et de ses paramètres de type)

```
template <class T> Point<T>::Point(T abs, T ord){
    x = abs;
    y = ord;
}
template <class T> Point<T> Point<T>::operator+(Point<T> oper){
    Point<T> p(x + oper.x, y + oper.y);
    return p;
}
```

Les fonctions membres *Point* et *operator+* sont des patrons de fonctions (d'où le mot clé *template*). *Point<T>::* indique que ces fonctions sont membres du patron de classe *Point* pour le type paramétré *T* (défini lors de l'appel).

Utilisation d'un patron de classe

- La définition de la classe sera instanciée en fonction des types fournis
  - Ces types devront être compatibles avec le patron de classe

```
Point<int> p1(0, 0);
Point<float> p2(1.5, 2);
```

Ici, deux instances de la classe *Point* sont créées. Dans la première, la paramètre de type est remplacé par *int*, et dans la seconde par *float*.

- Les définitions des fonctions membres sont des déclarations
  - Elles doivent être placées dans le fichier d'en-tête du patron de classe

Paramètres de type et d'expression

- Un patron de classe peut comporter des paramètres d'expression
  - Leurs valeurs sont précisées lors de l'instanciation d'une classe
- Les paramètres peuvent être en nombre quelconque
  - Ils peuvent être utilisés comme bon vous semble dans la définition du patron

```
template <class T, int s> class Vect{
    private:
        float tab[s];
    public:
        T operator[](int);
};
```

```
template <class T, int s> T Vect<T,s>::operator[](int i){
        if(0 <= i && i < s)
            return tab[i];
        else
            return tab[0];
}
```

Instanciation d'une classe patron

- Se déclare en fournissant ses arguments effectifs
  - Possible d'utiliser des paramètres de type instanciés par des patrons de classes
  - Pour les paramètres expressions, les paramètres effectifs doivent être constants
    - Le type doit être rigoureusement identique à celui prévu (aux conversions triviales près)

```
Vect<int, 5> v;
Point<Point<float>> p;
```

Ici, v est un vecteur d'entiers de taille 5 (voir slide précédent). p est un *Point* dont les coordonnées sont des *Points* à coordonnées réelles.

- Un patron de classe peut comporter des membres statiques
  - Chaque instance de la classe dispose de son propre jeu de membres statiques
- L'opération d'affection est définie pour chaque instance de la classe
  - Pas défini par défaut pour deux instances différentes

Exemple

TD 6
Exercice 3 et 4

```
template <class T> class Point{
    private:
        T x,y;
        static int nbr;
    public:
        Point(T, T);
        ~Point();
};
int Point<int>::nbr = 0;
int Point<float>::nbr = 0;
```

```
template <class T> Point<T>::Point(T abs, T ord){
    x = abs;
    y = ord;
    nbr++;
}
Point::~Point(){
    nbr--;
}
Point<int> p1(1, 2), p2(2, 3);
Point<float> p3(2.5, 3.5);
~~p1 = p3;~~
```

Point<int> et *Point<float>* correspondent à deux classes distinctes provenant d'un même patron. C'est pour cela qu'il est impossible d'affecter *p1* à *p3* qui sont de classes différentes (sauf en redéfinissant l' opérateur =)

Spécialisation de fonctions membres

- Il n'est pas possible de surdéfinir un patron de classe
  - Pas plusieurs patrons de même nom avec une liste de paramètres différents
- Il est néanmoins possible de spécialiser un patron de classe
  - Peut spécialiser des fonctions membres sans modifier la définition de la classe
  - Peut spécialiser la classe en fournissant une nouvelle définition
    - Dans ce cas, les fonctions membres peuvent être également spécialisées ou pas
  - Il est possible de prévoir des spécialisations partielles de patron de classe

Exemple

```
template <class T, int s> class Vect{
    private:
        T tab[s];
    public:
        void affiche();
};
template<class T, int s> void Vect<T,s>::affiche(){
    for(int i = 0; i < s; i++)
        cout << tab[i] << '\n';
}
void Vect <int, 2>::affiche{
    cout << tab[0] << " et " << tab[1] << '\n';
}
```

```
class Point{
    private:
        int x, y;
    public:
        Point(int = 0, int = 0);
        void affiche();
};
void Point::affiche(){
    cout << '(' << x << ',' << y << " ) \n ";
}
template<int s> void Vect<Point,s>::affiche(){
    for(int i = 0; i < s; i++)
        tab[i].affiche();
}
```

Ici, la fonction membre *affiche* est surdéfinie dans le cas de vecteurs d'entiers de taille 2, et dans le cas où les éléments du vecteur sont des *Points* (pour une taille quelconque).

Exemple

```
template <int s> class Vect<char,s>{
    private:
        char tab[s+1];
    public:
        Vect(char*);
        void affiche();
};
```

```
template <int s> Vect<char,s>::Vect(char *c){
        strncpy(tab, c, s);
        tab[s]='\0',
}
template <int s> void Vect<char,s>::affiche(){
        cout << tab << '\n';
}
```

Ici, la classe *Vect* est surdéfinie dans le cas où les éléments du tableau sont des *char* (pour que ce tableau soit considéré comme une chaine de caractères). Cette classe possède un constructeur qui prend en paramètre la chaine de caractère initiale. De plus, la classe surdéfinie la fonction membre *affiche*.

Paramètres par défaut

- On peut spécifier des paramètres par défaut pour certains paramètres

```
template < int s = 10 , class T = int> class Vect{
    private:
        T tab[s];
    public:
        void affiche();
};
Vect<5> v1;
Vect<> v2;
```

Le premier paramètre (d'expression) a pour valeur par défaut 10, et le second paramètre (de type) a pour valeur par défaut *int*. Le vecteur v1 est donc un vecteur de 5 entiers, alors que le vecteur v2 est un vecteur de dix entiers. Notez que deux classes distinctes sont créées.

Patrons de fonction membre

- Le mécanisme de patron de fonction s'applique aux fonctions membres
  - Peut s'appliquer à des fonctions membres de patrons de classes

```
template <class T, int s> class Vect{
    private:
        T tab[s];
    public:
        template<class U> &Vect<T,s> operator+(U);
};
template<class U> template<class T, int s> &Vect<T,s> Vect<T,s>::operator+(U operande){
    for(int i = 0; i < s; i++)
        tab[i] += operande;
    return *this;
}
```

Ici, *operator+* est une un patron de fonction dont le paramètre de type est celui de l'argument. Dans sa définition apparaissent également les paramètres de type et d'expression du patron de classe *Vect*. *v1+3* crée une fonction membre *operateur+* avec un paramètre de type *int*. L'opération + suivante, qui s'applique entre un *Point* et un flottant, crée également une nouvelle fonction membre.

```
Vect<int,10> v1, v2;
v2 = v1 + 3 + 3.5;
```

Déclarations d'amitié

- La déclaration peut porter sur un patron ou sur une instance particulière

```
template <class T> class Point;
template <class T, int s> class Vect{
    private:
        Point<T> tab[s];
    public:
        Point<T> sum();
};
template <class T> class Point{
    private:
        T x, y;
    public:
        template <int s> friend class Vect<T, s>;
};
```

```
template <class T, int s> Point<T> Vect<T,s>::sum(){
    Point<T> res = tab[0];
    for(int i = 1; i < s; i++){
        res.x += tab[i].x;
        res.y += tab[i].y;
    }
    return res;
}
```

Le patron de classe *Vect* permet de modéliser des vecteurs de *Points* dont le type des coordonnées est quelconque. Une déclaration d'amitié avec le patron de classe *Vect* apparait dans la classe *Point*, et toutes les fonctions membres de la classe *Vect* ont accès aux membres privés de la classe *Point*.