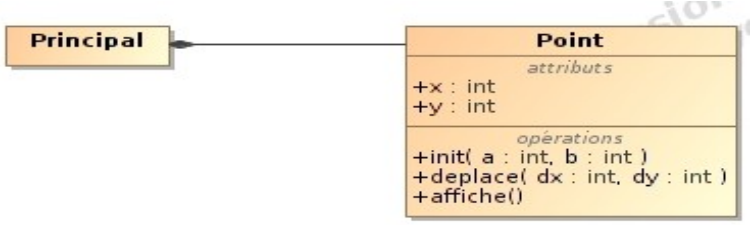
PROGRAMMATION OBJET LANGAGE CPP

1) LA NOTION DE CLASSE .

Nous allons enfin parler, dans ce chapitre, de Programmation Orientée Objet. Nous allons commencer par comprendre le mécanisme des classes.

1.1) Écriture d'une première classe

Une classe est en quelque sorte une structure complexe qui permet l'encapsulation de données. Une classe est composée de données et de méthodes. Lorsque l'encapsulation des données est parfaite, seules certaines méthodes sont accessibles, devenant ainsi l'interface. Ceci évite en principe à l'utilisateur de la classe, de se soucier de son fonctionnement et de faire des erreurs en changeant directement la valeur de certaines données.
Prenons un exemple concret et simple : l'écriture d'une classe Point. Cet exemple va nous suivre tout au long de ce chapitre.
La structure UML de cette classe est la suivante :



En C, nous aurions fait une structure comme suit :

```
struct Point
{
    int x; // Abscisse du point
    int y; // Ordonnée
};
```

La déclaration précédente fonctionne parfaitement en C++. Mais nous aimerions rajouter des fonctions qui sont fortement liées à ces données, comme l'affichage d'un point, son déplacement, etc.
Voici une solution en C++ :

```
class Point
{
public :
    int x;
    int y;
    void Init(int, int); // Initialisation d'un point
    void Deplace(int, int); // Déplacement du point
    void Affiche(); // Affichage du point
};
```

Vous remarquerez tout de suite plusieurs éléments :

- class : le "struct" a été remplacé, même si dans cet exemple précis, il aurait pu être conservé. Mais nous ne rentrerons pas dans les détails.
- public : le terme public signifie que tous les membres qui suivent (données comme méthodes) sont accessibles depuis l'extérieur de la classe. Nous verrons les différentes possibilités plus tard.
- L'ajout des fonctions (ou plutôt méthodes puisqu'elles font partie de la classe)

"Init", "Deplace" et "Affiche". Elles permettent respectivement d'initialiser un point, de le déplacer (addition de coordonnées) et de l'afficher (contenu des variables x et y).

1.2) Utilisation de la classe Point

Voici maintenant un programme complet pour mettre en application tout ceci :

Programme point.h

```
#include <iostream>
using namespace std ;
class Point
{
public :
    int x;
    int y;
    void Init(int a, int b) ;
    void Deplace(int a, int b) ;
    void Affiche() ;
};
```

Programme point.cpp

```
void Point::Init(int a, int b)
{
x = a;
y = b;
}
void Point::Deplace(int a, int b)
{
x += a;
y += b;
}
void Point::Affiche()
{
   cout << this->x<<" , " <<this->y<< endl;
}
```

Programme testpoint.cpp

```
int main()
{
    Point p;
    p.Init(3,4);
    p.Affiche();
    p.Deplace(4,6);
    p.Affiche();
    return 0 ;
}
```

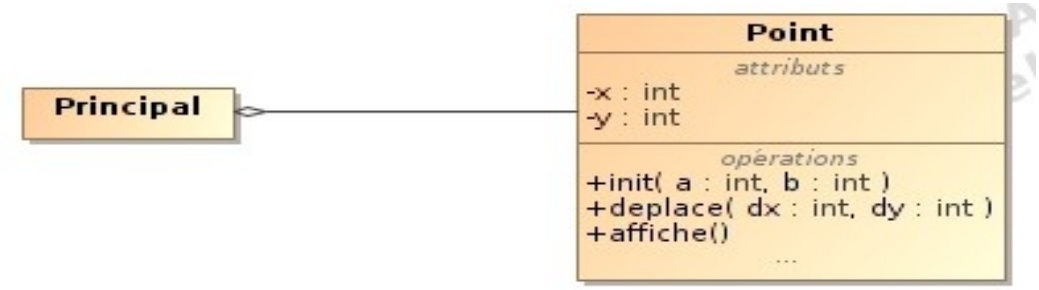
Tester cette application pour ceci vous réaliserez 3 fichiers

- **un fichier pour le programme principal : testpoint.cpp**
- **un fichier point.h où sera défini la classe Point.**
- **Un fichier point.cpp où seront écrite toutes les méthodes de la classe Point**

Vous avez remarqué la présence du "Point::" qui signifie que la fonction est en fait une méthode de la classe Point. . Ceci signifie que chaque appel à la méthode sera remplacé dans l'exécutable,

par le code de la méthode en elle-même (un peu comme une macro en C). D'où un gain de temps certain, mais une augmentation de la taille du fichier en sortie3.
En outre, vous verrez par la suite que l'on sépare généralement la déclaration de la classe de son implémentation, dans deux fichiers différents. Or, les méthodes ainsi implémentées nécessitent une recompilation globale lorsqu'on les modifie dans le .h, ce qui peut être gênant à la longue...
Mais la différence entre une structure et une classe n'a pas encore été vraiment détaillée. En effet, vous pourriez très bien compiler le même source en retirant le terme "public" et en remplaçant "class" par "struct".
En fait, l'intérêt réel du C++ tourne autour de cette notion importante d'encapsulation de données. Dans l'exemple de la classe Point, nous n'avons pour l'instant spécifié aucune protection de données ; vous pouvez rajouter ces quelques lignes, sans erreur de compilation :

1.3) Allocation dynamique des objets.



Il est conseillé d'allouer dynamiquement les objets en utilisant les opérateurs new et delete. Ceci permettra à la fermeture du programme de supprimer l'emplacement mémoire utilisé pour l'utilisation des objets du programme.

Modifier le programme précédent par celui-ci :

```
void main()
{
    Point *p=new Point;
    p->Init(3,4);
    p->Affiche();
    p->Deplace(4,6);
    p->Affiche();
    delete p ;
}
```

Rajouter ces lignes dans votre programme principal. Peut on atteindre les attributs x et y de l'objet p ?

```
...
p→x = 25; // Accès aux variables de la classe point
p →y = p→ x + 10;
cout << "le point est en " << p→x << ", " << p→y << endl;
p→affiche() ;
...
```

L'encapsulation a pour objet d'empêcher cela, afin de notamment limiter la nécessité de compréhension d'un objet par l'utilisateur. La classe devient alors une espèce de "boîte noire" avec des interfaces d'entrée et de sortie. D'où la déclaration suivante :

```
class Point
{
private :
    int x;
    int y;
```

```
public :
        void Init(int a, int b);
        void Deplace(int a, int b);
        void Affiche();
};
```

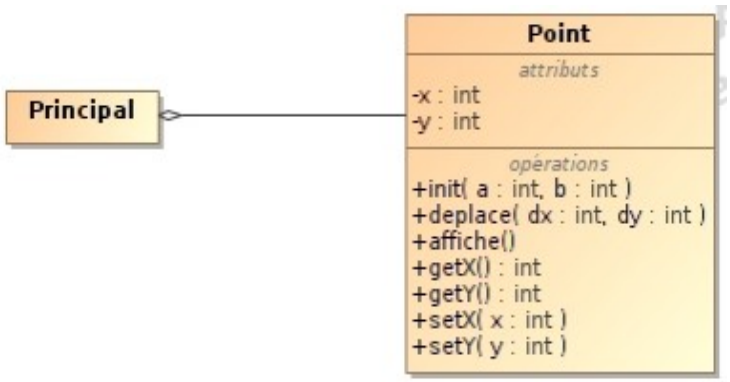
Il est alors interdit d'accéder aux variables x et y qui sont des membres "privés", en dehors de la classe Point (elles restent accessibles dans les méthodes de Point !). Il est également possible d'effectuer une affection de classe, comme pour une structure C. Ceci a le même effet puisque l'affectation a lieu sur les données membre :

```
#include <iostream>
... // déclaration de la classe point
void main()
{
        Point *p=new Point;
        p→Init(3,4);
        p→Affiche();
        delete p ;
}
```

Peut on atteindre les attributs x et y de l'objet p ? Justifier votre réponse.

1.4) Les Accesseurs et Mutateurs.

Pour accéder à ces variables privées on utilise généralement des accesseurs Get et des mutateurs Set.
La notation uml de la classe devient donc :



Le – signifie que les attribues x et y sont privées et le + que les méthodes sont publiques

Proposer un codage des accesseurs ci dessus définies dans la classe point getX, getY, setX, setY

Peut réaliser un accesseur pour les deux variables x et y simultanément. Si oui solution :
Peut réaliser un mutateur pour les deux variables x et y simultanément. Si oui solution :

1.5) Constructeur et Destructeur

Au cours de l'élaboration de cet exemple aussi simple que concret, vous vous êtes peut-être dit qu'il serait intéressant d'initialiser l'objet au moment de sa déclaration. En effet, il faut

de toute façon généralement utiliser tout de suite après la méthode "Init", alors pourquoi ne pas faire d'une pierre deux coups ! Ceci est bien entendu possible en C++ : il s'agit des constructeurs.
Voici comment nous pouvons mettre en œuvre un constructeur :

```
class Point
{
// Ici le "private" est optionnel dans la mesure où tout
// ce qui suit la première accolade est privé par défaut
        int x;
        int y;
public :
        Point(int, int); // Constructeur de la classe point
        void Init(int a, int b);
        void Deplace(int a, int b);
        void Affiche();
};
```

Vous vous demandez sûrement comment déclarer désormais, une variable de type Point !
Vous pensez peut-être pouvoir faire ceci : **Point *p=new Point;**

En fait, non. A partir du moment où un constructeur est défini, il doit pouvoir être appel par défaut pour la création de n'importe quel objet. Dans notre cas il faut par conséquent préciser les paramètres, par exemple :

```
Point *p=new Point(4,5);
```

Pour laisser plus de liberté, et permettre une déclaration sans initialisation, il faut prévoir un constructeur par défaut :

```
class Point
{
        int x;
        int y;
public :
        Point(); // Constructeur par défaut
        Point(int, int);
        void Init(int a, int b);
        void Deplace(int a, int b);
        void Affiche();
};
```

Dans notre cas de classe Point, le destructeur a peu d'utilité. On pourrait à la rigueur placer une instruction permettant de tracer la destruction.

1.6) Les Fonctions membres

1.6.1) Surdéfinition

Nous avons vu dans le chapitre précédent qu'il était possible de définir plusieurs constructeurs différents. Nous pouvons étendre cette possibilité de surdéfinition à d'autres méthodes que le constructeur (sauf le destructeur !) :

```
class Point
{
        int x;
        int y;
public :
      Point();
      Point(int, int);
      ~Point();
      void Init(int a, int b);
      void Init(int a); // Initialisation avec une même valeur
      void Deplace(int a, int b);
      void Deplace(int a);
      void Affiche();
      void Affiche(char* strMesg); // Affichage avec un message
};
```

1.6.2) Arguments par défaut

Tout comme une fonction C++ classique, il est possible de définir des arguments par défaut. Ceux-ci permettent à l'utilisateur de ne pas renseigner certains paramètres. Par exemple, imaginons que l'initialisation par défaut d'un point soit (0,0). Nous pouvons donc changer la méthode Init, de sorte qu'elle devienne :

```
void Init(int a=0);
```

Désormais, quand l'utilisateur appelle cette méthode, il a la possibilité de ne pas donner de paramètre, signifiant qu'il veut initialiser son point à 0. De même :
`void Affiche(char* strMesg="");`
Permet de remplacer l'implémentation de deux méthodes par une seule, mais qui prend en compte le non renseignement du paramètre. Notre programme devient donc :

```
class Point
{
        int x;
        int y;
public :
        Point();
        Point(int, int);
        ~Point();
        void Init(int a, int b);
        void Init(int a=0);
        void Deplace(int a, int b);
        void Deplace(int a=0);
        void Affiche(char* strMesg="");
};

Point::Point()
{
         cout << "--Constructeur par defaut--" << endl;
}

Point::Point(int a, int b)
{
        cout << "--Constructeur (a,b)--" << endl;
        Init(a,b);
}
```

```
Point::~Point()
{
        cout << "--Destructeur--" << endl;
}

void Point::Init(int a, int b)
{
           x = a;
           y = b;
}

void Point::Init(int a)
{
        Init(a,a);
}

void Point::Deplace(int a, int b)
{
            x += a;
            y += b;
}

void Point::Deplace(int a)
{
         Deplace(a,a);
}

void Point::Affiche(char *strMesg)
{
         // On ne rajoute pas le paramètre par
         // défaut dans l'implémentation !
         cout << strMesg << x << ", " << y << endl;
}

void main()
{
        Point *p=new Point(1,2);
        p->Deplace(4);
        p->Affiche("Le point vaut ");
        p->Init(10);
        p->Affiche("Le point vaut desormais : ");
        Point *pp=new Point;
        pp = p;
        p->Deplace(12,13);
        pp->Deplace(5);
        p->Affiche("Le point p vaut ");
        pp->Affiche("Le point pp vaut ");
        delete p ; delete pp ;
}
```

réaliser ce programme

Vous justifierez entr'autre la différence entre les appels suivants :

- **p->Deplace(12,13);**
- **et pp->Deplace(5);**

1.6.3) Objets transmis en argument d'une fonction membre .

Nous pouvons maintenant imaginer vouloir comparer deux points, afin de savoir s'ils sont égaux. Pour cela, nous allons mettre en oeuvre une méthode "Coïncide" qui renvoie "1" lorsque les coordonnées des deux points sont égales :

```
class Point
{
     int x;
     int y;
public :
     Point(int a, int b){ x=a; y=b; }
     int Coincide(Point p);
};

int Point::Coincide(Point  p)
{




}
```

Cette partie de programme fonctionne parfaitement, mais elle possède un inconvénient majeur : le passage de paramètre par valeur, ce qui implique une "duplication" de l'objet d'origine. Cela n'est bien sûr pas très efficace.
La solution qui vous vient à l'esprit dans un premier temps est probablement de passer par un pointeur. Cette solution est possible, mais n'est pas la meilleure, dans la mesure où nous savons fort bien que ces pointeurs sont toujours sources d'erreurs (lorsqu'ils sont non initialisés, par exemple). La vraie solution offerte par le C++ est de passer par des références. Avec ce type de passage de paramètre, aucune erreur est possible puisque l'objet à passer doit déjà exister (être instancié). En plus, les références offrent une simplification d'écriture, par rapport aux pointeurs :

```
#include <iostream.h>
class Point
{
      int x;
      int y;
public :
     Point(int a=0, int b=0) ;
     int Coincide(Point &);
};

int Point::Coincide(Point & p)
{


}

void main()
{
     Point *p=Point(2,0);
     Point *pp=Point(2);
```

```
    if( p->Coincide(*pp) )
        cout << "p et pp coincident !" << endl;
    if( pp->Coincide(*p) )
        cout << "pp et p coincident !" << endl;
}
```

2) CONSTRUCTION, DESTRUCTION ET INITIALISATION D'OBJETS

2.1) Constructeur par défaut/initialisant

2.1.1) Constructeurs simples

Nous avons abordé précédemment la possibilité d'initialiser un objet lors de sa construction, ou encore de proposer un constructeur par défaut. Nous allons dans ce chapitre généraliser le principe tout en l'approfondissant. Tout d'abord, il faut bien comprendre l'intérêt de la chose: un constructeur initialisant permet comme son nom l'indique d'initialiser les données membre, que ces valeurs soient entrées par l'utilisateur, ou bien qu'elles soient par défaut. Le destructeur quant à lui est appelé à la fin de la vie de l'objet, de façon automatique ou non (si l'objet est un pointeur).

Ceci prend toute son importance dans le cas d'un objet qui contient lui-même des données de type pointeur. Prenons par exemple une classe Vecteur qui gère un vecteur mémoire d'entiers. Nous allons essayer d'intégrer dans un premier temps :

- un constructeur initialisant, connaissant la taille du vecteur,
- un destructeur,
- une méthode de lecture d'une valeur dans le vecteur,
- une méthode d'écriture d'une valeur dans le vecteur,
- une méthode qui renvoie la taille du vecteur.

Dans la mesure où vous devriez être capable de réaliser cette classe, essayez de la concevoir tout seul dans un premier temps...

2.2) Le pourquoi du comment

Il faut bien comprendre le rôle du destructeur. Un destructeur de classe est appelé – comme son nom l'indique – à la destruction de l'objet, c'est-à-dire à la fin du bloc dans lequel il a été instancié, ou bien à un endroit spécifié par l'utilisateur, si l'objet a été créé dynamiquement (appel à un delete à ce moment là).

Mais le destructeur détruit seulement l'objet, soit la mémoire allouée pour les membres. En outre, il ne peut pas savoir que la classe est composée de pointeurs et qu'une allocation mémoire a été effectuée vers tel segment mémoire. C'est au créateur de la classe qu'il advient de spécifier la désallocation des pointeurs (tout comme l'allocation d'ailleurs).

Quant au main, il est vraiment très simple. Il se propose juste de créer un objet de type Vecteur, dynamiquement, de le remplir avec différentes valeurs, puis de les afficher. Pour comprendre un peu le fonctionnement de la classe, essayez d'autres opérations telles que différentes créations d'objets statiquement et dynamiquement.

2.3) Constructeur par recopie

Reprenons notre classe Point. Nous avons donc déjà deux constructeurs, à savoir un par défaut et un autre par initialisation des coordonnées. Il apparaît qu'il pourrait être intéressant de réaliser un constructeur à partir d'un point !

C++ permet cette fonctionnalité par défaut, en voici l'exemple :

```
// Définition de la classe Point
...
Point *pt=new Point(1,2);
Point *pt2=new Point(*pt); // Construction par recopie de Point
```

...
Il est possible de redéfinir ce constructeur. Bien entendu, il faut qu'il y ait un intérêt quelconque.

```
class Point
{
      int x;
      int y;
public :
      Point() ;
      Point(int a, int b) ;
      Point(Point & pt) ;// Constructeur par recopie
... // D'éventuelles autres méthodes
};
// Constructeur par recopie
Point::Point(Point & pt) ;
{


}
```

Ajouter un constructeur par recopie est donc assez simple dans l'esprit. En ce qui concerne la syntaxe, vous remarquerez deux choses :
- le passage par référence : C++ oblige un passage par référence dans le cas d'une construction par recopie. Ceci vient du fait qu'il faut réellement utiliser l'objet lui-même, et non une copie (d'ailleurs, le serpent se mordrait la queue sinon...).
- la présence d'un "const" : en fait, rien n'oblige de le placer ici, c'est simplement une protection de l'objet à recopier. En effet, lors de cette recopie, nous faisons appel à l'objet lui-même, et il n'y aurait aucun sens à vouloir le modifier !
Cet exemple est utile pour introduire la nécessité d'un constructeur par recopie. En effet, dans le cas d'un objet qui comporte un pointeur (notre classe Vecteur, par exemple), lorsqu'on veut recopier un objet, on ne recopie pas ce qui est pointé, mais l'adresse du pointeur seulement. Du coup, on se retrouve avec deux objets différents, mais qui possèdent une donnée qui pointe vers la même chose ! Ceci est bien entendu fort dangereux, et par là même, est à éviter.

Un petit schéma pour mieux comprendre : Figure 1 : recopie simple d'un objet Vecteur
On voit clairement qu'après recopie, les membres pVecteur de v et de v2 pointent au même endroit.
La parade à cela vient tout naturellement du constructeur par recopie. Au lieu de copier "bêtement" le pointeur, on peut allouer la mémoire et puis recopier les valeurs.

3) SURDÉFINITION D'OPÉRATEURS

3.1) Comment ça marche

Nous avons commencé à réaliser une classe Point qui permet la recopie d'objet. Mais jusqu'à maintenant, nous avons été obligés de créer des méthodes telles que "Coincide" qui vérifie que deux points sont égaux. Nous aurions pu également réaliser une fonction membre qui ajoute un point à un autre :

```
class Point
{
    int x;
    int y;
public :
    Point();
    Point(int a, int b);
    Point(const Point & pt);
    void Ajoute(const Point & pt);
    ... // D'éventuelles autres méthodes
};
```

Vous sentez alors qu'il serait très appréciable et plus naturel de pouvoir faire quelque chose comme :

Point a(1,2); Point b(3,4); Point c; c = a + b;	Point *a=new Point(1,2) ; Point *b=new Point(3,4) ; Point *c=new Point ; (*c)= (*a) + (*b) ;

C++ permet de réaliser une surdéfinition des opérateurs de base, comme " +", "-", "*", "/", "&", "^", etc. La liberté étant totale, vous pouvez faire réellement ce que vous voulez, par exemple une soustraction pour l'opérateur d'addition et inversement. Mais il est clair qu'il est plus que conseillé de respecter la signification de chacun de ces opérateurs.

3.2) Opérateurs simples

Mettre en oeuvre les opérateurs simples est une opération assez rapide. Un exemple est utile pour vous montrer comment on fait :

```
/* point.h */
class Point
{
    int x;
    int y;
public :
    Point();
    Point(int a, int b);
    Point(const Point & pt);
    Point operator+(const Point & a) ;

    ... // D'éventuelles autres méthodes
};

/*point.cpp*/
Point Point::operator+( const Point & a)
{
Point p;
```

```
p.x = a.x + x;
p.y = a.y + y;
return p;
}
```

Des explications sont nécessaires. Tout d'abord, vous remarquerez le "const Point & a". Ceci signifie que l'on passe un point en paramètre (l'autre point de l'addition est en fait la classe appelante elle-même). Ce dernier est transmis par référence, afin d'éviter une recopie, lourde et moins rapide. Le "const" est optionnel mais permet d'éviter de modifier les paramètres et également autorise l'utilisation d'objets constants.
L'opérateur rend un Point. En fait, on rend en fin de méthode le point qui a été créé temporairement au départ et qui contient la somme des deux paramètres. Plus exactement, on rend une copie de cet objet, le retour de la fonction étant "Point", et non "Point&" ou "Point*". Ceci est normal. Il faut savoir que l'objet créé dans la méthode sera automatiquement détruit à la fin de cette dernière. On ne peut bien évidemment pas rendre l'adresse d'un objet qui sera détruit ! D'où la nécessité d'une recopie.
Ce que nous avons mis en oeuvre pour l'opérateur d'addition, nous pouvons en faire de même pour tous les autres opérateurs "simples". Quelques exemples :

```
class Point
{
     int x;
     int y;
public :
     …….
     Point operator +=(const Point & a);
     Point operator +(const Point & a);
     bool operator==(const Point & p);
     ……..
};

Point Point::operator +=(const Point & a)
{ // Addition de 2 points

}

Point Point::operator +(const Point & a)
{ // Soustraction de 2 points

}

bool Point::operator==(const Point & p)
{ // Egalité de 2 points (remplace "Coincide")


}
```

Compléter les fonctions ci dessus permettant de créer les opérateurs +, -, ==, += et -= et réaliser le test à partir des fonctions ci dessous.

```
int main()
{
   Point *p=new Point(7,7);
   p->Affiche();
   Point *pp=new Point(7,7);
   pp->Affiche();
   if ((*pp)==(*p))
      cout<<"points identiques"<<endl;
   else cout <<"points differents"<<endl;;
   Point *p1=new Point();
   p1->Affiche();
  // (*p1)=(*pp);
  // p1->Affiche();
   (*p1)=(*pp)+(*p);
   pp->Affiche();
   p->Affiche();
   p1->Affiche();
   (*p1)+=(*p);
   p1->Affiche();
   return 0;
}
```

3.3) Opérateur d'affectation

L'opérateur d'affectation "=" représente un cas particulier. En effet, nous retrouvons le même problème que lors de la construction par recopie : il est toujours possible d'effectuer une affectation entre deux objets (de même type), mais que se passe-t-il s'ils contiennent des pointeurs (cf. problématique de recopie) ! C'est pourquoi il est souvent important d'implémenter ce type d'opérateur.
Il n'est pas plus difficile à mettre en oeuvre :

Cette fois-ci, nous rendons par contre une référence sur l'objet, car nous devons rendre la classe elle-même et non une copie, comme vous pouvez le comprendre.

Exercice :
Réaliser et tester l'opérateur d'affectation pour la classe point.

4) L'HÉRITAGE

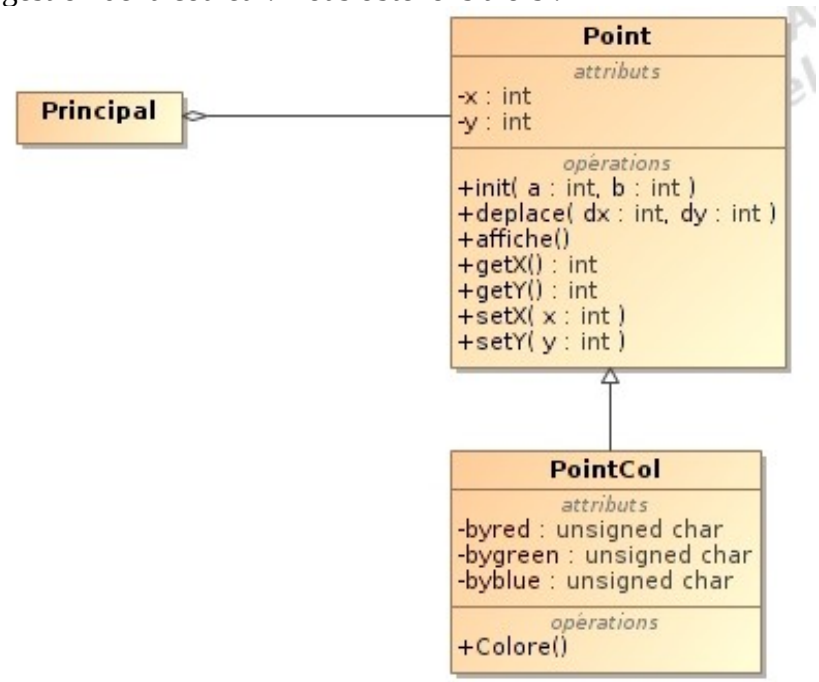
4.1) Définition et mise en oeuvre

4.1.1) Définition

Nous avons vu dans le premier chapitre que l'héritage est en C++, et plus généralement en P.O.O., un concept fondamental. En effet, il permet de définir une nouvelle classe "fille", qui héritera des caractéristiques de la classe de base. Le terme "caractéristiques" inclut en fait l'ensemble de la définition de cette classe mère.
D'un point de vue pratique, il faut savoir qu'il n'est pas nécessaire à la classe fille de connaître l'implémentation de la base : sa définition suffit. Ceci sera mis en application. De plus, on peut hériter plusieurs fois de la même classe, et une classe fille pourra également servir de base pour une autre. Il est alors possible de décrire un véritable "arbre d'héritage".

4.1.1) Mise en œuvre

Mettre en oeuvre la technique de l'héritage est assez en simple en C++. Le plus difficile reste en fait la conception, qui nécessite un gros travail afin de bien séparer les différentes classes.
Le premier exemple qui nous permettra de réaliser notre premier héritage, propose de définir une classe PointCol qui hérite de Point. Sémantiquement parlant, PointCol est un Point auquel on rajoute la gestion de la couleur. Nous obtenons alors :




```
class PointCol : public Point
{
    unsigned char byred;     // La composante rouge
    unsigned char bygreen; // La composante verte
    unsigned char byblue;   // La composante bleue
public :
    // Coloration d'un point
    void Colore(unsigned char R,unsigned char G,unsigned char B )
    {
        byred = R;
        bygreen = G;
        byblue = B;
    }
};
```

Vous pouvez remarquer la notation ": public Point". Ceci signifie que notre point coloré hérite publiquement de Point, c'est-à-dire que tous les membres publics de Point seront membres publics de PointCol.
En déclarant un objet de type PointCol, il est ainsi possible d'accéder aux membres publics de PointCol, donc, mais également de Point. C'est une notion essentielle de la P.O.O. !
Pour mettre en application notre exemple, nous allons utiliser la classe Point qui suit. Nous allons pour la première fois faire cela dans les "règles de l'art", en séparant physiquement la définition de l'implémentation (un fichier ".h" et un fichier ".cpp").

Reprendre les fichiers associés au Point réalisé précédemment.

```
#include "Point.h"
class PointCol : public Point
{
    unsigned char byred;      // La composante rouge
    unsigned char bygreen;    // La composante verte
    unsigned char byblue;     // La composante bleue
public :
// Coloration d'un point
void colore(unsigned char r, unsigned char g, unsigned char b)
    { // Programme à écrire dans le fichier PointCol.cpp
        byred = r;
        bygreen = g;
        byblue = b;
    }
};
```

Vous pouvez maintenant rajouter le fichier main.cpp suivant :

```
void main()
{
    PointCol *ptc= new PointCol();
    ptc->colore( 64, 128, 192 );
    ptc->Affiche();
    ptc->Deplace( 3, 6 );
    ptc->Affiche();
}
```

4.2) Utilisation des membres de la classe de base

Utiliser des membres de la classe de base est simple à réaliser. Il faut cependant faire attention à leur statut. Les membres privés ne peuvent être appelés. Soit une méthode "InitCol" qui initialise un point coloré :

```
void InitCol( int Abs, int Ord, unsigned char r, unsigned char g, unsigned char b )
{
    Point::Init(Abs, Ord);
    byred =r;
    bygreen = g;
    byblue = b;
}
```

Il suffit donc d'appeler la méthode souhaitée, précédée de la classe.

4.3) **Redéfinition des fonctions membre et appel des constructeurs**

L'appel à la méthode "Affiche" fonctionne très bien, et utilise en fait la déclaration faite dans la classe Point, ce qui est logique puisque PointCol n'en possède aucune autre. Mais que se passe-t-il si on veut afficher un point coloré ?

Une première solution consiste à introduire une nouvelle méthode "AfficheCol" dans la classe fille. En plus de cette méthode, nous allons ajouter un constructeur qui permet l'initialisation de la classe PointCol. Vous voyez immédiatement quelle pourrait être la définition :

PointCol(int Abs, int Ord, unsigned char R, unsigned char G, unsigned char B);

Il contient donc les coordonnées du point, plus les composantes de la couleur. La mise en oeuvre est un peu plus complexe. Soit notre classe Point :

```
class Point
{
     int x;
     int y;
public :
     Point(int a=0, int b=0);
     //……...
};
```

Nous voyons bien que, quelque part, il faudrait passer les coordonnées entrées en paramètres du constructeur initialisant de PointCol, vers celui de Point ! Ceci s'effectue de la façon suivante :

```
#include "Point.h"
class PointCol : public Point
{
     unsigned char byred;
     unsigned char bygreen;
     unsigned char byblue;
public :
     PointCol(int,int,unsigned char,unsigned char,unsigned char);
     void colore( unsigned char, unsigned char, unsigned char );
};



// Constructeur initialisant de la classe PointCol,
// faisant appel au constructeur initialisant de Point.
PointCol::PointCol( int Abs, int Ord, unsigned char r, unsigned  char g, unsigned char b) :
Point(Abs, Ord)
{
     byred = r;
     bygreen = g;
     byblue = b;
}

void PointCol::Colore( unsigned char r, unsigned char g, unsigned char b )
{
     byRed = r;
     byGreen = g;
```

```
    byBlue = b;
}
```

Remarquez la transmission de paramètres effectuée par le ":Point(Abs, Ord)". C'est en fait un appel au constructeur initialisant de la classe de base. Il est possible d'étendre cette mise en oeuvre à tous les constructeurs, par exemple par recopie.
Essayez ! Vous pouvez également changer le type d'héritage et le rendre "private", pour voir la différence.
Dans certains cas, il peut être intéressant de pouvoir avoir accès aux données membres de la classe de base. Par exemple, reprenons notre affichage dans PointCol :

```
#include "Point.h"
class PointCol : public Point
{
    unsigned char byred;
    unsigned char bygreen;
    unsigned char byblue;
public :
    PointCol(int,int,unsigned char,unsigned char,unsigned char);
    void colore( unsigned char, unsigned char, unsigned char );
    void AfficheCol();
};

PointCol::PointCol( int Abs, int Ord, unsigned char r, unsigned char g, unsigned char b) :
Point(Abs, Ord)
{
    byRed = r;
    byGreen = g;
    byBlue = b;
}
void PointCol::Colore( unsigned char r, unsigned char g, unsigned char b )
{
    byred = r;
    bygreen = g;
    byblue = b;
}

void PointCol::AfficheCol()
{
    Point::Affiche();
    // Notez le "cast" en "int" des composantes nécessaire, sinon
    // le compilateur prend les valeurs (char) pour des caractères
cout << "Couleur : RGB(" << (int)byred << "," << (int)bygreen<< "," << (int)byblue << ")."
<< endl; ;

}
```

Le résultat est satisfaisant, mais un appel à la méthode Affiche de Point est peut-être fastidieux, d'autant plus qu'il pourrait être intéressant d'avoir accès aux coordonnées du point, directement. Ceci n'est pas possible ! Si vous essayez, le compilateur vous donnera une erreur de type :
cannot access private member declared in class „Point". Ceci vient du fait que les membres x et y de Point sont privés.

4.4) "Statuts" de dérivation

La solution à ce problème permet de laisser les membres inaccessibles aux utilisateurs de la classe, mais pas des classes qui en héritent. Il suffit de remplacer le "private" par "protected". Notre classe Point devient alors :

```
class Point
{
protected :
      int x;
      int y;
public :
      Point(int abs, int ord){ x=abs; y=ord; }
      void Deplace(int a, int b){ x+=a; y+=b; }
      void Affiche();
};
```

Et notre méthode AfficheCol :
Cela permet de gagner du temps également à l'exécution, par rapport à un appel de fonction par exemple...

```
void PointCol::AfficheCol()
{
      cout << "Point (" << x << ", "<< y << ") ";
      cout << "de couleur : RGB("<<(int)byRed<<","<<(int)byGreen;
      cout << "," << (int)byBlue << ")." << endl;
}
```

L'intérêt du statut protégé est donc double puisque les données se retrouvent inaccessibles pour l'extérieur, ce qui préserve l'encapsulation des données de la classe, mais par contre demeurent toujours utilisables pour d'éventuels héritages.

4.5) Mode de dérivation

Pour déclarer une classe B dérivée d'une classe de base A, on utilise la syntaxe suivante :

```
class B : modeDérivation A
{
     Membres de la classe B
}
```

modeDeDérivation peut être :

- public : la protection des membres de la classe A (classe de base) reste inchangée au niveau de la classe B (classe dérivée de A).
- protected : les membres publics et protected de la classe A sont considérés comme protected dans la classe B. Dans ce cas les classes dérivées de B ont toujours accès aux membres de la classe A.
- private : les membres publics et protected de la classe A sont considérés comme private dans la classe B. Dans ce cas les classes dérivées de B n'ont plus accès aux membres de la classe A.

Un membre privé d'une classe est définitivement inaccessible pour l'extérieur de la classe, c'est à dire pour les objets de cette classe, pour les fonctions membres des classes dérivées et pour les objets des classes dérivées.
Les tableaux ci-dessous résument les propriétés des membres public, private et protected de la classe de base pour les différents modes de dérivation.

Mode de dérivation public

Statut des	Les membres de la sont-il	Les membres de la classe	Statut des membres

membres de la classe de base	accessibles classe de base par les fonctions membres de la classe dérivée ?	de base sont-il accessibles par les objets de la classe dérivée ?	de la classe de base dans la classe dérivée
public	oui	oui	public
protected	oui	non	protected
private	non	non	Inaccessible

Mode de dérivation protected

Statut des Les membres de la classe de base	Les membres de la de base sont-il accessibles classe par les fonctions membres de la classe dérivée ?	Les membres de la classe de base sont-il accessibles par les objets de la classe dérivée ?	Statut des membres de la classe de base dans la classe dérivée
public	oui	non	protected
protected	oui	non	protected
private	non	non	Inaccessible

Mode de dérivation private

Statut des Les membres de la classe de base	Les membres de la de base sont-il accessibles classe de base par les fonctions membres de la classe dérivée ?	Les membres de la classe de base sont-il accessibles par les objets de la classe dérivée ?	Statut des membres de la classe de base dans la classe dérivée
public	oui	non	private
protected	oui	non	private
private	non	non	Inaccessible

4.6) Notion d'héritage : élargissement

Il est possible d'étendre la notion d'héritage à plusieurs classes : un véritable arbre peut-être créé, par exemple une classe C qui hériterait de A et B (héritage multiple). Nous n'aborderons pas ces fonctionnalités dans ce cours, mais vous pouvez consulter un livre plus complet qui abordera certainement cela.

Voir le cours sur UML.

5) FONCTIONS VIRTUELLES

5.1) Utilité

Nous avons acquis dans le chapitre précédent, la notion d'héritage. Elle nous permet en outre de créer de véritables arbres de classes. Reprenons notre exemple de Point et de PointCol. Nous avons implémenté des méthodes qui permettent l'affichage, ou encore l'initialisation des données, dans chacune des deux classes : Affiche dans Point, AfficheCol dans PointCol par exemple.
Je suppose que vous vous êtes demandé pourquoi nous ne leur avons pas donné le même nom ! Le mieux pour le comprendre est d'essayer.

```
... // Définition de la classe PointCol identique...
void PointCol::Affiche ()
{
    cout << "Point (" << x << ", "<< y << ") ";
    cout << "de couleur : RGB("<<(int)byRed<<","<<(int)byGreen;
    cout << "," << (int)byBlue << ")." << endl;
}
```

Cette déclaration de la classe PointCol à l'exécution, donne les résultats voulus, à savoir que c'est la "bonne" méthode Affiche qui est appelée. En fait, la liaison est établie statiquement à la compilation. Ici, le compilateur sait très bien quelle fonction membre utiliser. Mais maintenant, imaginons l'utilisation suivante :

```
void main()
{
      PointCol ptc(5,10, 50,150,200);
      ptc.Affiche();
      Point pt(52,17);
      pt.Affiche();
      Point *ppt; // Pointeur de point "générique"
      ppt = &ptc;
      ppt->Affiche(); // le pointeur pointe désormais sur un point coloré : légal !
}
```

Le résultat peut vous sembler surprenant. En fait, le typage étant effectué statiquement, pour le compilateur, ppt reste quoi qu'il advienne un pointeur sur Point. Or, nous n'avions pas encore vu cela, mais il est possible d'affecter une adresse de classe fille à un pointeur de classe de base...
Dans ce cas, vu que l'affectation est dynamique, un appel de la méthode Affiche utilise en fait la déclaration de la classe de base, Point.
Un autre exemple pour illustrer l'utilité des fonctions virtuelles, consisterait à réaliser un affichage "descendant". Il s'agit de réaliser un affichage de toutes les informations relatives aux deux classes, Point et PointCol, en appelant une seule méthode de Point. Vous comprendrez mieux cela, en étudiant le code :

```
// Point.h
class Point
{
     int x;
     int y;
public :
     Point(int a, int b){ x=a; y=b; }
     void Init(int a, int b){ x=a; y=b; }
```

On appelle cela polymorphisme, et permet de passer par exemple un pointeur sur la classe de

base, en quelque sorte "générique", lorsqu'on utilise plusieurs classes différentes qui en héritent.

```
    void Deplace(int a, int b){ x+=a; y+=b; }
    void Affiche();
    void AfficheTout();
};

// Point.cpp
#include "Point.h"
void Point::Affiche()
{ cout << this << "->" << x << ", " << y << endl; }

void Point::AfficheTout()
{
    cout << this << "->" << x << ", " << y << endl;
    Affiche();
}

// PointCol.h
#include "Point.h"
class PointCol : public Point
{
    unsigned char byRed;
    unsigned char byGreen;
    unsigned char byBlue;
public :
    PointCol(int,int,unsigned char,unsigned char,unsigned char);
    void Colore( unsigned char, unsigned char, unsigned char );
    void Affiche();
};

PointCol::PointCol( int Abs, int Ord, unsigned char R, unsigned char G, unsigned char B) :
Point(Abs, Ord)
{
    byRed = R;
    byGreen = G;
    byBlue = B;
}

void PointCol::Colore( unsigned char R, unsigned char G, unsigned char B )
{
    byRed = R;
    byGreen = G;
    byBlue = B;
}

void PointCol::Affiche()
{
    cout << "Couleur : RGB(" << (int)byRed << "," << (int)byGreen;
    cout << "," << (int)byBlue << ")." << endl;
}
```

En appelant AfficheTout dans le programme, nous souhaitons afficher les renseignements de la classe Point (code contenu dans la première ligne de la méthode), mais aussi ceux de la classe

appelante. Par exemple :

```
void main()
{
    PointCol ptc(5,10, 50,150,200);
    ptc.Affiche();
    Point pt(52,17);
    pt.Affiche();
    ptc.AfficheTout();
}
```

Dans ce cas précis, nous affichons les informations de couleurs de ptc, puis de coordonnées de pt. La dernière ligne devrait afficher les informations de coordonnées et de couleurs de ptc. Mais dans ce premier test, il n'en est rien !

5.2) Mécanisme

Dans le paragraphe précédent, nous avons vu que dans certaines conditions, donner le même nom à une méthode de la classe fille qu'une méthode de la classe de base, peut être source d'erreur, ou plutôt, d'incompréhension.
Pour éviter cela, il faudrait pouvoir indiquer au compilateur de lier certaines méthodes dynamiquement. En effet, il y a des cas où, seulement à l'exécution, le programme peut savoir quelle fonction membre employer. Les fonctions virtuelles servent à cela. Soit la définition suivante :

```
// Point.h
class Point
{
    int x;
    int y;
public :
    Point(int a, int b){ x=a; y=b; }
    void Init(int a, int b){ x=a; y=b; }
    void Deplace(int a, int b){ x+=a; y+=b; }
    virtual void Affiche();
    void AfficheTout();
};

// Point.cpp
void Point::Affiche()
{ cout << this << "->" << x << ", " << y << endl; }

void Point::AfficheTout()
{
    cout << this << "->" << x << ", " << y << endl;
    Affiche();
}
```

Essayez maintenant les deux tests précédemment implémentés. Vous constaterez que grâce à ce virtual, la liaison est désormais dynamique et donc, que la bonne méthode est appelée.

5.3) Remarque

Les fonctions virtuelles permettent ainsi de mettre en œuvre un processus très intéressant de

liaison dynamique, en ce sens que la bonne méthode à utiliser est sélectionnée à l'exécution. Cette souplesse est couramment utilisée lorsque l'on se retrouve avec de nombreuses classes héritées, et que l'on passe par exemple en paramètre un pointeur sur la classe mère.
Cette facilité a un coût : elle est très gourmande ! En effet, la liaison dynamique est assez lourde, et il convient d'en disposer avec parcimonie. De ce fait, il est parfois fortement conseillé de concevoir son programme sans utiliser ce processus, par exemple au sein d'un traitement d'image, qui est déjà suffisamment gourmand...

5.4) Identification de type à l'exécution

Pour information, et sans rentrer dans les détails, il est intéressant de savoir que le C++ a introduit au cours de ses évolutions, la possibilité de connaître le type d'une variable (identification, comparaison), à l'exécution7. Cette possibilité fait partie de la fameuse Librairie Standard, que nous n'avons pas encore abordée, et qui fera l'objet du chapitre 10.
Voici un exemple très succinct :

```
#include <iostream>
using namespace std;
int n(10);
cout << typeid(n).name() << endl;
int nn(15);
if( typeid(n)==typeid(nn)
cout << "Meme type !" << endl;
```

La ligne "using namespace std;" permet de faire connaître au compilateur cette possibilité.
En bref, typeid(variable) renvoie un id de type, qui peut être comparé avec l'opérateur classique de comparaison. Il est également possible de connaître le nom du type, avec typeid(variable).name(), ou encore, de savoir si une variable est d'un type ascendant à celui d'une autre, à l'aide de typeid(variable).before(). Vous comprendrez l'intérêt sous-jacent, notamment en polymorphisme.
Ainsi :

```
#include
class A { int n; };

class B : public A { int nn; };

int main()
{
A a;
B b;
cout << typeid(a).before(typeid(b));
return 0;
}
```

Nous obtiendrons à l'affichage "1", puisque la variable a est bien d'une classe mère au type de la variable b.
Pour mieux comprendre le fonctionnement de la librairie standard, se référer à la partie 10.
De plus, il faut savoir que cette possibilité est essentiellement utilisée dans le cadre du développement des programmes, par exemple pour le débogage.
Ou R.T.T.I. pour Run Time Type Identification, en anglais dans le texte.

6) EXERCICE PRÉLIMINAIRE

Vous avez appris très rapidement les "bases" du C++ dans les précédents chapitres. Il est évident que vous n'avez pas encore pu assimiler toutes ces notions. Pour ce faire, rien n'est plus efficace qu'une mise en application. L'exercice qui vous est proposé, a pour objet la réalisation d'un ensemble de deux classes, qui s'occupent de la gestion de chaînes de caractères formatées (mise en italique, gras et couleur).
La première classe gère la première partie, à savoir la chaîne de caractères. En voici sa description rapide :
- gestion d'une chaîne de caractères et de sa taille,
- constructeur par défaut,
- constructeur initialisant à partir d'une chaîne de caractères (char*),
- destructeur,
- surcharge de l'opérateur = (affectation de chaîne),
- surcharge de l'opérateur == (égalité de chaîne),
- surcharge de l'opérateur += (trois différents, un qui gère un String, un autre un char* et un dernier un char),
- surcharge de l'opérateur + (concaténation),
- surcharge de l'opérateur [] (accès à un caractère de la chaîne stockée),
- vérification de l'initialisation de la classe (on vérifie que la chaîne n'est pas vide),
- mise à zéro des paramètres (chaîne vide),
- renvoie de la taille de la chaîne,
- et affichage de la chaîne.

Vous êtes bien entendu libre de rajouter d'autres méthodes.
La deuxième classe hérite de la première, et rajoute une couche gérant le formatage du texte.
Description sommaire :
- gestion du formatage : Italic, Bold et Couleur (un short pour faire simple),
- constructeur par défaut,
- constructeur connaissant une chaîne de caractères et éventuellement les options de formatage,
- constructeur connaissant une chaîne de caractères de type String et éventuellement les options de formatage,
- constructeur connaissant les options de formatage,
- constructeur par recopie,
- destructeur (optionnel),
- surcharge de l'opérateur = (affectation) pour le cas d'une copie de chaîne formatée,
- surcharge de l'opérateur = (affectation) pour le cas d'une copie de chaîne non formatée (classe de base),
- méthodes permet la gestion de l'italique (mise en "italic" et renvoi d'information),
- méthodes permet la gestion de Bold (mise en "Bold" et renvoi d'information),
- Colorisation et renvoie de couleur,
- Affichage de la chaîne et des informations de formatage, en utilisation la notation HTML, à savoir :

1. pour la mise en italique et à la fin,
2. pour la mise en gras et à la fin,
3. pour la couleur et à la fin.

La réalisation de ces classes est assez simple si vous procédez par étape. Utilisez également très allègrement les exemples fournis auparavant. N'hésitez pas à regarder comment on déclare un constructeur par recopie, etc.
C'est également le moment pour voir de plus près comment fonctionne le debugger, car vous ne serez pas sans faire quelques erreurs... de frappe !

BON COURAGE !