1. Introduction du cours de C++

Ce cours est une présentation du langage C++ dans le cadre de la programmation orientée objet. Le langage C++ est une surcouche du langage C. Il peut donc s'utiliser comme un « C amélioré » : en utilisant les flots d'entrée-sortie (cin/cout) à la place des fonctions scanf et printf, en utilisant des objets de la classe string plutôt que les chaînes de caractères du C, ou encore en utilisant les conteneurs vector de la bibilothèque STL à la place des tableaux statiques ou dynamiques. De plus, le C++ introduit la notion de référence (notion déjà présente en langage Pascal) qui facilite la gestion des paramètres des fonctions et rend obsolète le passage d'argument « par adresse ».

Mais la vocation du C++ est avant tout la mise en oeuvre des concepts de la Programmation Orientée Objet (POO). En POO, un programme ne repose pas seulement sur des appels de fonctions ; il décrit la coopération d'objets. Un objet est une variable structurée avec différents attributs, et l'accès aux attributs se fait par l'intermédiaire de fonctions membres, elles-mêmes attachées à l'objet. Les données (les valeurs des attributs) sont dites encapsulées dans les objets. Vu de l'extérieur, on ne connaît un objet que par ce qu'il sait faire, c'est-à-dire son comportement.

Cette approche de la programmation rend l'évolution des programmes plus facile. Par exemple, il est possible de changer l'implémentation d'un objet sans changer son comportement. On facilite ainsi le découplage de certaines parties du logiciel. En outre, les objets bien conçus sont facilement réutilisables dans d'autres contextes.

Pour ces raisons, la POO est très différente de la programmation procédurale. En POO, on ne cherche pas à décomposer un logiciel en fonctions et sous-fonctions. On cherche plutôt à identifier quelles sont les classes d'objets pertinentes pour un problème donné et quelles sont les relations entre ces classes. Pour appliquer les concepts de la POO, le C++ autorise la définition de classes (attributs, fonctions membres, visibilité des membres), et la mise en œuvre des relations (notamment de composition ou de spécialisation) entre les classes. De plus, des classes standards d'objets sont fournies avec les compilateurs C++: on peut utiliser des objets string pour gérer les chaînes de caractères et les classes stream (istream, ostream, fstream) permettent les entrées-sorties de type clavier/écran/fichiers.

L'objectif de ce cours de C++ est de présenter l'écriture de classes en C++, la façon d'implémenter la composition (voire l'agrégation), ainsi que l'implémentation de la spécialisation (notion de sous-classe). Pour illustrer ces notions, des exemples simples seront donnés. Comme pour tout langage informatique, il convient de comprendre et retenir un certain nombre de techniques et de règles d'écriture justifiées par des critères de performance, ou de validité du code. Des fiches synthétiques résumant les points importants sont fournies.

2. Le C++ en dehors des classes d'objets

Le langage C++ a été conçu par Bjarne Stroustrup au début des années 1980. Le C++ est une surcouche au langage C qui intègre les concepts de la POO. Le C++ préserve donc à de minimes exceptions la syntaxe du langage C. L'objectif de B. Stroustrup en créant ce langage était de faciliter aux programmeurs C la transition vers la programmation orientée objet.

2.1. Les notions du C utilisées en C++

Le C++ est un langage typé. Le compilateur C++ vérifie donc la compatibilité des types lors des opérations arithmétiques, des affectations et des passages de paramètres aux fonctions.

2.1.1. Les types primitifs (ou types de base).

Les types primitifs du C++ sont les mêmes qu'en C, à savoir `char, short, int, long, float, double`, les variables non signées `unsigned char, unsigned short, unsigned int` et les variables pointeur

`char*, short*, int*, double*` etc. Le nombre d'octets assigné à chacun de ces types peut varier d'un compilateur à un autre.

Rappel : la taille d'une variable (exprimée comme multiple de la taille d'un `char`) s'obtient à l'aide de l'opérateur `sizeof`.

```
int var;
printf("taille de var=%d, taille d'un char= %d",sizeof(var),sizeof(char));
```

2.1.2. Les pointeurs

Les *variables pointeurs* sont toujours utilisées en C++ notamment pour le passage de tableaux en paramètre d'une fonction. Rappelons qu'un pointeur représente un type.

Une variable de type `T*` est destinée à recevoir l'adresse d'une variable de type `T`.

Lorsque `ptr` est une variable pointeur de type `T*`, l'opérateur `*` permet de *dé-référencer* ce pointeur : `*ptr` est la variable de type `T` dont l'adresse est stockée dans la variable `ptr`.

Il est à noter que `ptr` est une variable et `*ptr` en est une autre. Par ailleurs, l'opérateur `&` permet de fournir l'adresse d'une variable. En synthèse on a :

```
int var=12;    // variable de type int déclarée et initialisée
int * ptr;     // variable de type pointeur sur int non initialisée.

// ne pas exécuter *ptr=45 ici, la variable pointeur n'est pas initialisée !!!

ptr=&var ;     // ptr initialisée avec l'adresse de la variable var.
*ptr=45 ;      // *ptr dé-référence ptr. A cet instant, *ptr est la variable var.
```

Rappel : puisque les pointeurs sont typés, le compilateur réalise aussi des vérifications de type impliquant les pointeurs :

```
int varI=12;
int * ptrI=&varI;
char varC=12;
char* ptrC=&varC;
ptrC=&varI ;   // erreur de compilation : char * non compatible avec int *
ptrI=&varC ;   // idem : int * pas compatible avec char *
```

2.1.3. Les pointeurs et les tableaux.

En C et en C++, le nom d'un tableau correspond à l'adresse de sa première case. De plus, on peut toujours utiliser une variable pointeur comme s'il s'agissait de l'adresse de la première case d'un tableau :

```
int tab[3]={1,2,7};   // tableau de 3 int initialisés
int * ptrI=tab ;      // licite, identique à ptrI=&tab[0]
ptrI[2]=3 ;           // licite : ptrI[2] équivaut ici à tab[2]
ptrI=&tab[1] ;        // licite aussi , identique à ptrI=tab+1
ptrI[1]=2 ;           // ptrI[1] équivaut maintenant à tab[2]
```

Rappelons que les chaînes de caractères en C sont aussi des tableaux:

```
char chaine2[26]="bonjour";  // tableau de 26 char dont 8 cases sont initialisées
char chaine2[]="bonjour";    // tableau de 8 char initialisés
// surtout pas char * chaine3 ="bonjour";
```

2.1.4. Les structures de contrôle.

Le C++ utilise la même syntaxe que le langage C pour les différentes structures de contrôle. Les structures de contrôle sont if/else, for et while (ou do ... while).

La structure **`if(…){…}else{…}`**

```
int entier;
printf("Saisir un entier:\n");
scanf("%d",&entier);

if((entier%2)==0)
{
      printf("Entier pair");
}
else
{
      printf("Entier impair");
}
```

La structure **`while(…){…}`** ou **`do{…}while(…);`**

```
int entier;
do
{
      printf("Saisir un entier impair :\n");
      scanf("%d",&entier);
}
while((entier%2)==0);       //boucle tant que entier pair
```

La structure **`for(… ;… ;…){…}`**

```
int tab[10];  // réserve un tableau de 10 entiers (non initialisés)
int indice;
tab[0]=0;           // initialise le premier élément à 0

for(indice=1;indice<10;indice++)
{
      tab[indice]=tab[indice-1]+indice;    // tab[indice]=indice + (indice-1)+...
}
```

2.1.5. Les conversions de type (transtypage ou cast)

Le compilateur C++ utilise et vérifie les types pour s'assurer que les opérations réalisées sont cohérentes et que les paramètres passés à une fonction correspondent aux paramètres attendus. Néanmoins, dans certains cas, le compilateur autorise des conversions *implicites* (conversions que le compilateur peut déduire grâce au contexte) ou *explicites* de types. Il faut noter que certaines conversions implicites génèrent tout de même des mises en garde (warnings) au moment de la compilation (par exemple lorsque l'on compare un entier signé avec un entier non signé).

Il y a par exemple conversion implicite quand on affecte la valeur d'une variable de type float à une variable de type int (ou l'inverse). Dans cette situation, le compilateur met en place automatiquement une conversion de type. Cette conversion est dite *dégradante* (perte d'information) dans le sens float-> int. L'exemple ci-dessous présente quelques conversions de type.

```
int varI=513;         // variable entière sur 32 bits signé
char varC;            // variable entière sur 8 bits signé
float varF=2.5 ;      // variable utilisant la norme flottant IEEE 32 bits

varC=varI;     // conversion dégradante
varI=varF;     // ici, un warning signale « possible loss of data »
printf("%d %d",varC,varI);   // affiche 1 (513mod256) et 2(partie entière de 2.5)
```

Parfois, les incompatibilités de type conduisent à l'échec de la compilation :

```
int varI=513;
char * ptrC=&varI;    // conversion int* -> char * impossible.
```

Lorsque le programmeur souhaite outrepasser les incompatibilités de type, œlui-ci peut indiquer explicitement la conversion à réaliser. On parle alors de conversion explicite, de transtypage explicite ou encore de cast. Dans ce cas, le programmeur a la responsabilité de s'assurer de la validité de l'opération.

```
int varI=513;
char * ptrC=(char *) &varI;          // conversion int*-> char* demandée
printf("%d %d",ptrC[0],ptrC[1]);     // affiche les deux octets de poids faible de varI
```

Lors des conversions explicites (comme ci-dessus), c'est au programmeur de vérifier scrupuleusement ce qu'il fait. Cet aspect du langage donne beaucoup de liberté au programmeur et permet de gérer des problèmes de bas niveau. Mais l'abus de conversions explicites, notamment pour régler des warnings récalcitrants à la compilation, peut conduire à de grosses erreurs de programmation. L'exemple suivant illustre ce qu'il ne faut pas faire même si le compilateur l'autorise.

```
int varI=513;
char * ptrC=(char *) &varI;          // cast int * -> char *
int adresse=(int) ptrC;              // cast char * -> int (ça c'est vraiment douteux)
int * pI=(int *) adresse;            // cast int -> int *  (aussi douteux)
*pI=2;                               // miraculeusement, pI contient ici l'adresse de varI.
printf("%d",varI);
```

Les conversions de pointeurs sont parfois utiles pour tromper le compilateur. En revanche les conversions de pointeurs en valeurs numériques (ou l'inverse) sont farfelues. L'exemple précédent montre que le compilateur accepte tout et n'importe quoi dès qu'on utilise les conversions de type.

Rappelons quelques cast utiles :

```
T * -> V* : conversion de pointeurs. Les adresses restent intactes car elles ont la même
taille. Mais ces adresses ne désignent pas des variables de même type ! A utiliser avec
précaution.
float -> int : (dégradant) conserve la partie entière.
int -> float : passage au format flottant IEEE
char -> int : la variable passe sans altération de 8 bits à 32 bits
int -> char : dégradant : les 32 bits sont tronqués aux 8 bits de poids faible.
```

2.1.6. Les fonctions et le passage de paramètres en C (et en C++)

En C++, on utilise abondamment les fonctions au sein des classes (on parlera de fonctions membres ou de méthodes d'une classe). Il est donc nécessaire de rappeler brièvement le mode de transmission des paramètres.

En C et en C++, les paramètres sont passés à une fonction *par copie*, c'est-à-dire via la pile. Le programme qui utilise une fonction recopie dans la pile les paramètres d'appel (encore appelés paramètres effectifs). Par conséquent, la fonction appelée n'a accès (dans la pile) qu'à une *copie* des paramètres effectifs, et non à l'original. Il faut noter que les variables locales à une fonction sont également des emplacements mémoire alloués dans la pile (comme les paramètres). En résumé, qu'il s'agisse des paramètres ou des variables locales à une fonction, ce sont des variables *temporaires* qui n'existent que le temps d'exécution de la fonction.

```
void MaFonction(int param)           // param ppv pAppel
{
       param++ ;      // ici, param est modifié mais pas pAppel
}
```

```
void main()
{
	int pAppel=23;
	MaFonction(pAppel);		// pAppel est recopié dans param
	printf("d",pAppel);	// pAppel vaut toujours 23
}
```

Vu différemment, on peut considérer qu'au début de la fonction le paramètre formel (`param`) est une variable locale initialisée à l'aide de la valeur du paramètre d'appel, mais qu'il s'agit bien de deux variables distinctes. Ce mode de passage est appelé *passage par copie* ou *passage par valeur*.

Par conséquent, lorsque l'on souhaite qu'une fonction modifie une variable, on n'a pas d'autre choix que de lui fournir l'adresse de la variable à modifier. On appelle souvent cette technique *passage de paramètres par adresse*.

```
void MaFonction(int * adrVariable)
{
	(*adrVariable)++;
}

void main()
{
	int pAppel=23;
	int * adrAppel=&pAppel;
	MaFonction(adrAppel) ;
	printf("pAppel =%d",pAppel); //pAppel vaut ici 24
}
```

Remarquez bien que cette fois-ci c'est *l'adresse qui est passée par copie*, mais on peut désormais modifier la variable désignée par l'adresse contenue dans `adrAppel`, peu importe que la variable pointeur `adrAppel` ne puisse pas être modifiée par la fonction !

2.1.7. Performances liées au mode de transmission des paramètres

Le mode de transmission des paramètres peut avoir des répercussions sur la rapidité d'exécution. Considérons le programme suivant où un type structuré `personne` est défini. Une variable de type `personne` contient 3 champs : deux tableaux de 30 caractères et un entier non signé. Une variable de type personne a une taille de 64 (60 char + 1 unsigned codé sur 32 bits).

Rappel : pour les pointeurs sur des variables structurées, la notation `ptr->nom` est équivalente à `(*ptr).nom`

```
--------------------------------------------------------------------------------
struct personne
{
	char nom[30];
	char prenom[30];
	unsigned age;
};

void Affiche(personne p)
{
	printf(" nom : %s prenom : %s age : %d \n",p.nom,p.prenom,p.age);
}

void AfficheBis(personne * ptr)
{
	printf(" nom : %s prenom : %s age : %d \n",ptr->nom,ptr->prenom,ptr->age);
}
```

```
void main(void)
{
	personne AM={"Martin","Arthur",32};
	printf("sizeof(AM) = %d\n",sizeof(AM)); // affiche : sizeof(AM)=64

	Affiche(AM);

	AfficheBis(&AM);	// exécution plus rapide que Affiche(AM)
}
--------------------------------------------------------------------------------
```

La fonction `AfficheBis(personne *)` est plus rapide que la fonction `Affiche(personne)` pour un service équivalent. Pourquoi ? L'appel de la fonction `Affiche(personne)` nécessite une recopie d'une variable de 64 octets dans la pile, alors que l'appel de la fonction `AfficheBis(personne *)` ne requiert que la copie d'un pointeur de 32 bits (4 octets).

Les soucis de performance imposent au programmeur de bien connaître tous ces aspects afin de réaliser les meilleurs choix. On verra qu'en C++ on utilisera le passage de référence plutôt que le passage d'adresse pour améliorer les performances.

2.1.8. Les paramètres constants : garantie d'un code avec moins de bugs

```
float Division(int num,int den)
{
	if(den=0) return num;
	return (float)num/den ;
}
```

La fonction précédente contient une erreur de programmation que le compilateur ne peut détecter. En effet, `den=0` est une affectation et non un test d'égalité. Le paramètre `den` prend donc systématiquement la valeur 0. Comment prévenir de telles erreurs ? On peut utiliser le modificateur const pour les paramètres d'entrée (paramètres qui ne sont pas censés évoluer dans la fonction).

```
float Division(const int num,const int den)
{
	if(den=0) return num;		// error C2166: l-value specifies const object
	else return (float)num/den ;
}
```

Dans ce cas, le compilateur indique qu'une variable (ou un objet) constante est à gauche d'une affectation (l-value). Le compilateur détecte ainsi très facilement ce type d'erreur. Le programmeur qui prend l'habitude de déclarer comme constants les paramètres d'entrée prévient ce type d'erreur.

2.2. Les références en C++

L'opérateur `&` ci-dessous correspond à la définition d'une référence (où `T` est un type).

```
T var;
T & ref = variable;	// ref est une référence à var
```

Ci-dessus, `ref` est une *référence* à la variable `var` de type `T`.[1] Cela signifie que les identificateurs `var` et `ref` désignent la même variable. Manipuler l'un, revient à manipuler l'autre. Dès lors, `var` et `ref` sont des synonymes, ou des *alias*. L'initialisation d'une référence lors de sa déclaration est obligatoire.

[1] Attention à ne pas confondre l'emploi de l'opérateur `&` pour obtenir l'adresse d'une variable et pour déclarer une référence !

```
--------------------------------------------------------------------------------
#include <iostream.h>
void main(void)
{
       int I=3;
       int &refI = I;

       printf("I = %d \n",I);
       printf("refI = %d \n",refI);

       printf("adresse de I = %x \n",&I);
       printf("adresse de refI = %x \n",&refI);

       printf("Incrément de I (I++)");
       I++;
       printf("refI = %d \n",refI);
}
--------------------------------------------------------------------------------
```

Résultats

```
I = 3
refI = 3
adresse de I : 0065FDF4
adresse de refI : 0065FDF4
Incrément de I (I++)
refI = 4
```

On constate que les variables `I` et `refI` ont la même adresse**.** Elles désignent donc bien la même variable. Puisque `I` et `refI` sont des alias, modifier l'un revient à modifier l'autre**.**

2.2.1. Les références en C++ : un nouveau mode de transmission de paramètres

L'utilisation des références faite précédemment ne présente pas d'intérêt particulier. L'utilisation la plus courante en C++ concerne l'échange d'informations avec une fonction. Cette partie du cours est essentielle et doit faire l'objet d'une attention toute particulière La notion de référence va nous affranchir, dans beaucoup de cas, de la technique de passage par adresse rappelée précédemment.

```
#include <stdio.h>

       void f(int &); // noter que le pramètre est une référence

void main(void)
{
       int pAppel=5;
       f(pAppel);
       printf("pAppel = %d \n",pAppel);
}

void f(int & pFormel)// pFormel est un alias de pAppel
{
       pFormel++;     // ici on  modifie réellement le paramètre d'appel, grâce à l'alias
}
```

Résultats

```
pAppel = 6
```

Ici le paramètre formel `pFormel` de la fonction `f(int & )` est une référence au paramètre d'appel. Dans ce cas, manipuler le paramètre formel `pFormel` revient à manipuler le paramètre d'appel. Par la suite, on appellera ce mode de transmission *passage par référence*

Le passage d'argument par référence autorise donc la *modification* du paramètre d'appel au sein de la fonction. Ce mode de passage convient donc aux *paramètres de sortie* et aux *paramètres d'entrée/sortie*. De plus, l'utilisation d'une référence est moins lourde que la technique du passage par adresse.

En terme de performances, le passage par référence est équivalent au passage par adresse. Lors de l'appel de la fonction, seule l'adresse du paramètre passé par référence est transmise à la fonction. Le compilateur gère les références grâce à des pointeurs qui sont cachés au programmeur.

2.2.2. Les paramètres passés par référence constante : efficacité et robustesse

Le passage par référence est aussi performant que le passage par adresse. Il évite ainsi certaines duplications d'objets volumineux (nécessitant beaucoup d'octets en mémoire). Du coup, cela expose aussi le programmeur à des erreurs de programmation qui se répercutent sur le programme appelant. Reprenons l'exemple d'une fonction ayant un paramètre d'entrée dont la taille en mémoire est importante. Les différentes fonctions illustrent les modes de passage de paramètres possibles en C++.

```
-------------------------------------------------------------------------------
struct personne
{
       char nom[30];
       char prenom[30];
       unsigned age;
};
void PassageParValeur(personne p)
{
       /* inconvénient : duplication du paramètre d'appel (64 octets ici)*/
       printf(" nom : %s prenom : %s age : %d \n",p.nom,p.prenom,p.age);
}
void PassageParValeurConst(const personne p)
{
       printf(" nom : %s prenom : %s age : %d \n",p.nom,p.prenom,p.age);
}
void PassageParAdresse(personne * ptr)
{
       /* avantage : plus rapide que par copie
        inconvénient : les erreurs peuvent se propager au programme appelant */
       printf(" nom : %s prenom : %s age : %d \n",ptr->nom,ptr->prenom,ptr->age);
}
void PassageParAdresseConst(const personne * ptr)
{
       printf(" nom : %s prenom : %s age : %d \n",ptr->nom,ptr->prenom,ptr->age);
}
void PassageParReference(personne & p)
{
       /* avantage : plus rapide que par copie (aussi bien que par adresse)
       inconvénient : les erreurs peuvent se propager au programme appelant */
       printf(" nom : %s prenom : %s age : %d \n",p.nom,p.prenom,p.age);
}
void PassageParReferenceConst(const personne & p)
{
       /* mode de transmission privilégié en C++ pour les paramètres d'entrée */
       printf(" nom : %s prenom : %s age : %d \n",p.nom,p.prenom,p.age);
}
-------------------------------------------------------------------------------
```

2.2.3. Bilan sur le passage de paramètres en C++

On appelle *paramètre d'entrée* d'une fonction un paramètre dont la valeur est utilisée par la fonction pour réaliser un traitement mais dont la valeur n'est pas modifiée par la fonction. On appelle *paramètre de sortie* un paramètre dont la valeur en entrant dans la fonction est sans importance mais dont la valeur va être modifiée par la fonction. Enfin, certains paramètres peuvent être à la fois d'entrée et de sortie. En C++, d'après la nature entrée-sortie d'un paramètre et sa taille, on peut choisir le mode de passage de paramètre le plus adapté.

Les paramètres d'entrée : si peu volumineux (moins de 4 octets : char, int, float, long ...) utiliser le passage par valeur, sinon le passage par référence constante

Les paramètres de sortie et d'entrée/sortie : passage par référence (le passage par adresse est obsolète)

Remarque : les tableaux sont toujours passés aux fonctions sous la forme de deux paramètres : l'adresse de la première case puis la taille du tableau.

On donne ci-dessous quelques exemples de fonctions simples en C++.

Fonction **Incrementer(E/S param : entier)** permettant d'ajouter 1 au paramètre. Le paramètre est de type entrée-sortie (E/S).

```
void Incrementer(int & param){ param++ ;}
```

Fonction **PlusUn(E param :entier)** retournant la valeur du paramètre (entier) d'entrée augmentée de 1. Le paramètre n'est plus ici qu'un paramètre d'entrée.

```
int PlusUn(int param){return (param+1); }
```

Fonction **Echanger(E/S a:entier,b :entier)** permettant d'échanger les valeurs des deux paramètres. Les deux paramètres sont de type E/S.

```
void Echanger(int & a, int & b){
       int c=a ;
       a=b ;
       b=c ;
}
```

Fonction **Random(S r:entier)** générant une valeur aléatoire. Le paramètre est de type sortie S.

```
void Random(int & r) { r=rand(); }
```

2.3. Les paramètres avec valeurs par défaut

En C++, des valeurs par défaut peuvent être fournies à certains paramètres. Dans ce cas, si le paramètre d'appel fait défaut, le paramètre formel de la fonction prend pour valeur la *valeur par défaut*.

```
---------------------------------------------------------------------------
#include <stdio.h>

       float Addition(float, float =3);     // 2nd paramètre avec valeur par défaut

void main(void)
{
       printf("%f \n",Addition(10,5));
       printf("%f \n",Addition(7));
}

float Addition(float a, float b)
{
       return a+b;
}
---------------------------------------------------------------------------
```

Résultats

```
15
10
```

En cas d'absence d'un second paramètre d'appel pour la fonction `Addition()`, le second argument prend pour valeur 3.

Note : les arguments ayant des valeurs par défaut doivent tous figurer *en fin de liste des arguments*. Voici quelques exemples, bons et mauvais, de prototypes :

```
float Addition(float=2, float =3);          // OK
float Addition(float=2, float);             // NON !!!
float Addition(float=1, float, float =3);   // NON PLUS !!!
float Addition(float, float=5, float =3);   // OK
```

Remarque : la valeur qu'un paramètre doit prendre par défaut doit figurer *au niveau du prototype* mais pas au niveau de l'en-tête de la fonction.

2.4. La surcharge de fonctions

Définition (Signature) : on appelle *signature* d'une fonction la combinaison de sa classe (si elle est membre d'une classe) de son identificateur et de la suite des types de ses paramètres. Les 3 fonctions suivantes ont des signatures différentes :

```
float Addition(float);
float Addition(int,float);
float Addition(float,float );
```

Note : le type de la valeur de retour n'appartient pas à la signature.

En C++, il est possible de définir plusieurs fonctions avec le même identificateur (même nom) pour peu qu'elles aient des signatures différentes. On appelle cela *la surcharge de fonction*.

L'exemple suivant présente un programme où deux fonctions `Addition()` sont définies. Il faut noter qu'il s'agit bien de fonctions différentes et qu'elles ne réalisent donc pas le même traitement. Le compilateur réalise l'appel de la fonction appropriée à partir du type des paramètres d'appel.

```
-------------------------------------------------------------------------------
#include<stdio.h>
      int Addition(int, int=4);
      float Addition(float, float =3);

void main(void)
{
      float fA=2.5;
      float fB=3.7;
      int iA=2;
      int iB=3;
      printf("%f \n",Addition(fA,fB));
      printf("%i \n",Addition(iA,iB));
      printf("%f \n",Addition(fA));
      printf("%d \n",Addition(iA));
}

float Addition(float a, float b){   return a+b; }

int Addition(int a,int b){ const int offset=12; return a+b+offset; }
-------------------------------------------------------------------------------
```

Résultats

```
6.2
17
5.5
18
```

3. Introduction à la POO et à la représentation de classes en UML

On va tenter de décrire brièvement ce qui distingue la POO (Programmation Orientée Objet) de la programmation procédurale (C, Pascal).

Les langages *procéduraux* s'appuient sur les capacités de la machine en terme de découpage en instructions élémentaires et sous-programmes (fonctions) (utilisation de CALL/RET pour les processeurs Intel). La programmation en langage procédural reste donc très liée aux caractéristiques des processeurs. Dans ce

contexte, pour réaliser un logiciel, on fait une analyse fonctionnelle descendante pour mettre en évidence les fonctions principales et les structures de données manipulées par ces fonctions. On découpe le problème initial en sous-problèmes, autant de fois qu'il nous paraît nécessaire, jusqu'à pouvoir résoudre les sous-problèmes par des fonctions de « taille raisonnable ». En pratique, cette approche présente de gros désavantages dans les projets informatiques. D'une part les fonctions écrites spécialement pour un projet sont rarement utilisables dans un autre projet. D'autre part, une modification des structures de données entraîne de multiples points de correction du logiciel. Les logiciels conçus avec cette approche évoluent difficilement et leur maintenance n'est pas aisée.

Les classes d'objets

La Programmation Orientée Objet (POO) cherche à s'adapter de manière plus naturelle à notre perception de la "réalité". Dans notre quotidien, nous réalisons naturellement une *classification* de ce qui nous entoure. Nous mettons chaque élément de notre entourage dans une *classe* d'éléments qui ont des caractéristiques communes fortes. Un élément d'une classe sera appelé de manière générique *objet* (même s'il s'agit d'un être vivant) par la suite. Les êtres vivants, les végétaux, les minéraux, les véhicules, les téléphones, les ordinateurs, les étudiants, les systèmes électroniques, les meubles sont autant d'exemples de classes d'objets (on rappelle qu'objet prend ici un sens large). Une classe décrit donc un ensemble d'objets.

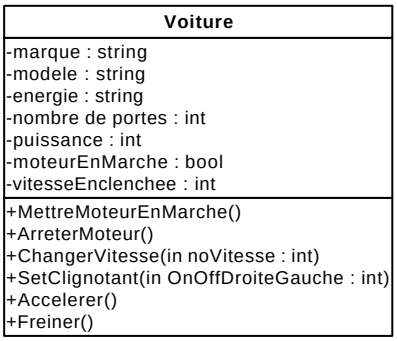
Les classes ne sont pas nécessairement disjointes. Par exemple, la classe des êtres humains est une sous-classe des êtres vivants, les hommes/les femmes sont deux sous-classes des êtres humains, les animaux une sous-classe des êtres vivants, les chiens une sous-classe des animaux etc.

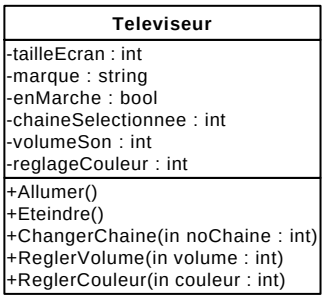
Attributs, comportement, état

Dans chacune des classes, les objets sont caractérisés par des *attributs* et des *comportements*. La valeur des attributs à un instant donné caractérise *l'état* d'un objet. Le comportement d'un objet peut avoir des conséquences sur son état et inversement, l'état d'un objet peut contraindre ses comportements possibles (à un instant donné).

Objet de la classe des téléviseurs
attributs : taille d'écran, marque, état allumé/éteint, chaîne sélectionnée, volume sonore ...
comportement : allumer, éteindre, changer de chaîne, régler le volume sonore, ...
Objet de la classe des voitures
attributs : marque, modèle, énergie (essence/diesel), puissance, nombre de portes ...
comportement : mettre moteur en marche, l'arrêter, changer de vitesse, ...

Dans la notation UML (Unified Modeling Language), les classes sont représentées par des rectangles indiquant quels sont les attributs, leur visibilité (- privé et + public), quelles sont les méthodes accessibles.





Enumérer les attributs et comportements d'un objet n'est pas toujours chose possible. On imagine mal dans un programme de gestion de comptes bancaires, décrire un client propriétaire d'un compte par l'ensemble de ses caractéristiques morphologiques ou psychologiques. Sans doute qu'on se contentera de le

décrire par son état civil plus quelques autres renseignements. Toute représentation d'un objet ou d'une classe d'objets nécessitera d'isoler les attributs et les comportements pertinents pour l'utilisation que l'on fera de cet objet. Cette démarche d'abstraction est arbitraire et est dépendante du contexte d'utilisation de la classe d'objets. En informatique, une classe d'objets va décrire quels sont les attributs et les comportements retenus pour décrire un objet au sein d'un programme.

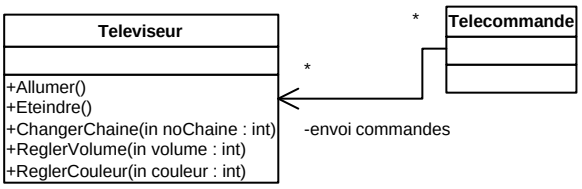
Encapsulation des données

Un objet contient des données et des méthodes prévues pour accéder à ces données. L'idée de la POO est de ne pouvoir "dialoguer" avec un objet qu'à travers l'interface constituée par l'ensemble de ses méthodes. Les méthodes jouent donc le rôle d'interface entre les données et l'utilisateur de l'objet. Elles filtrent les accès aux données. Pour assurer cette encapsulation, des mot-clés définissent la visibilité des membres d'une classe.

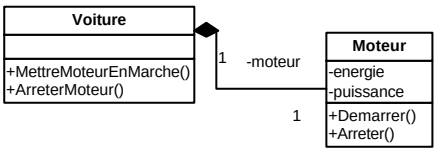
Les relations entre les objets (par extension, entre les classes)

L'idée de la POO est d'utiliser nos aptitudes naturelles à classifier des objets et à établir des liens entre eux. Dès lors, un logiciel va être le résultat de l'interaction d'objets entre eux. Encore faudra-t-il, avant de faire collaborer des objets entre eux, établir quelles sont les classes d'objets pertinentes , et quelles sont leurs relations.

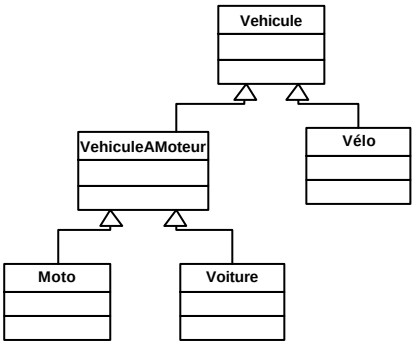
Il est à noter que différentes relations peuvent apparaître entre des objets ou entre des classes. Deux objets peuvent par exemple s'échanger des données. Par exemple, un objet télécommande communique avec un objet téléviseur...



Deux objets peuvent être en relation du type *composant /composite*. Un objet voiture est *composé* d'un objet moteur. La notation UML introduit un lien avec un losange du côté du composite.



Enfin, les différentes classes peuvent être en relation hiérarchique. On peut décrire une classe comme représentant un sous-ensemble d'une classe plus générale. La relation entre une classe et une sous-classe est une relation de type *spécialisation/généralisation*. La classe des vélos est une sous-classe des véhicules. La classe des véhicules à moteur est une sous-classe des véhicules. Dans ce sens, un véhicule à moteur est plus spécifique qu'un véhicule, ou inversement, un véhicule est plus général qu'un véhicule à moteur. La flèche part d'une sous-classe (ou classe spécialisée) et va vers une classe plus générale (ou super-classe).



Un programme en POO contiendra donc l'écriture des différentes classes d'objets et leur utilisation. Chaque classe sera définie par des attributs (accessibles seulement par les méthodes de la classe), et par des fonctions membres (ou méthodes). Enfin, les relations entre les classes devront également être implémentées. On verra notamment que la relation de généralisation peut être naturellement réalisée en C++. Parfois, pour une même relation, plusieurs réalisations différentes peuvent être implémentées en C++.

4. Les classes en C++

Dans un premier temps, on va s'attacher à l'écriture des classes s'appuyant seulement sur des types primitifs. Ceci va nous permettre d'introduire la syntaxe et les automatismes à acquérir pour des classes plus complexes.

4.1. Les classes ostream et istream

Avant d'aller plus loin, notons qu'en C++ des classes standard d'entrée-sortie facilitent la saisie et l'affichage d'informations. Les entrées sorties standard s'effectuent grâce aux objets `cin` (entrée standard) et `cout` (sortie standard). Ces objets sont des instances des classes `istream` (flot d'entrée) et `ostream` (flot de sortie). Ces classes sont définies dans le fichier d'entête `iostream` (sans .h).

Les objets `cin` et `cout` permettent l'affichage et la saisie de tous les types de base `char, int, float` … et même les chaînes `char *`.

```
-------------------------------------------------------------------------------------
#include<iostream>   // définition des classes ostream et istream
using namespace std; // espace de nom utilisé pour ces classes (explication ultérieure)

void main()
{
       char chaine[10]="Toto";
       int age=12;
       cout << chaine << " a " << age << " ans.";
       cout << "Saisir un entier : \n";
       cin >> age;
       cout << "Valeur saisie = " << age;
}
-------------------------------------------------------------------------------------
```

4.2. Méthodes et attributs

Le C++ introduit le mot clé `class` pour définir une classe d'objets. Une classe est un type structuré avec des champs de données typés (attributs) et des fonctions membres (méthodes). Pour expliquer la syntaxe, le plus simple est encore de donner un exemple. La classe ci-après est une classe de points à coordonnées entières. La définition de la classe est située dans un fichier d'entête (point.h), la définition des méthodes dans un fichier source (point.cpp) et enfin l'utilisation dans un autre fichier source (main.cpp). Les mots clé public et private indiquent la visibilité des membres. Les membres privés ne sont accessibles que par les objets de la classe. Par convention, les attributs sont toujours à accès privé.

```
-------------------------------------------------------------------------------
// point.h  fichier d'entête contenant la définition de la classe

#include<iostream>
 #ifndef __POINT__
 #define __POINT__
 class Point
 {
 public:              //partie publique de la classe
       void SetX(const unsigned x);                // les méthodes
       void SetY(const unsigned y);
       void SetXY(const unsigned x,const unsigned y);
```

```
        unsigned GetX() const;
        unsigned GetY() const;
        float Distance(const Point & p) const;
        void Translate(const unsigned x,const unsigned y);

        void Display(std::ostream & flot) const;

 private:      // partie privée (encapsulée)
        unsigned _x;                              // les attributs
        unsigned _y;
 };     // <- ne pas oublier ce point virgule
 #endif
 ------------------------------------------------------------------------------
```

Le fichier d'entête contient la définition de la classe. Celle-ci décrit les données membres (attributs) et les méthodes. En fait, bien que ce ne soit pas nécessaire, seuls les prototypes des fonctions membres (méthodes) sont généralement donnés dans la définition de la classe.

```
------------------------------------------------------------------------------
// point.cpp  fichier source contenant la définition des fonctions membres
#include "point.h"   //inclusion du fichier d'entête de la classe
#include<math.h>

void Point::SetX(const unsigned int x)
{
       _x=x;
}

void Point::SetY(const unsigned int y)
{
       _y=y;
}

void Point::SetXY(const unsigned int x,const unsigned int y)
{
       _x=x;
       _y=y;
}

unsigned Point::GetX() const{
       return _x;
}

unsigned Point::GetY() const{
       return _y;
}

void Point::Translate(const unsigned Dx, const unsigned Dy)
{
       _x=_x+Dx;
       _y=_y+Dy;
}

float Point::Distance(const Point & p) const
{
       return sqrt((_x-p._x)*(_x-p._x)+(_y-p._y)*(_y-p._y));
}

void Point::Display(std::ostream & flot) const
{
       // affichage sur flot de sortie
       flot << "(" << _x << "," << _y << ")";
}
 ------------------------------------------------------------------------------
```

Le fichier d'implémentation des méthodes (ici `point.cpp`) contient le code des méthodes de la classe. Il faut noter la syntaxe permettant de définir une fonction membre. Le nom de la classe propriétaire de la méthode précède le nom de la méthode. Enfin, l'utilisation de la classe est faite dans un autre fichier.

```
-------------------------------------------------------------------------------
// main.cpp : fichier d'utilisation de la classe point
#include "Point.h"    // inclusion de la définition de la classe Point
using namespace std;

void main()
{
       Point p1,p2;          // p1 et p2 sont deux objets de la classe Point

       p1.SetXY(10,10);      // donne l'état (10,10) à l'objet p1
       p2.SetXY(20,20);      // donne l'état (20,20) à l'objet p2

       p1.Display(cout);     // affiche p1 sur le flot de sortie
       p2.Display(cout);     // affiche p2 sur le flot de sortie

       cout << "Distance p1 - p2 :" ;
       cout << p1.Distance(p2) << endl;     //affiche la distance p1-p2

       // p1._x=12 ; est interdit par le compilateur, les données sont privées
}
-------------------------------------------------------------------------------
```

Que faut-il remarquer ?

- les champs privés ne sont pas accessibles par l'utilisateur de la classe
- les attributs privés sont néanmoins accessibles aux objets de la classe, c'est-à-dire dans la définition des méthodes (regarder le code de la méthode distance)
- deux objets de la même classe ont les mêmes attributs mais pas nécessairement le même état

4.3. Constructeurs : méthodes d'initialisation d'un objet

Il ne manque pas grand chose pour que la classe `Point` définie précédemment soit réellement utilisable. Son plus gros défaut est que l'état d'un objet de la classe `Point` est indéfini avant l'utilisation de la méthode `Point::SetXY()`. Le C++ prévoit un mécanisme pour réaliser une initialisation correcte des attributs dès la création d'un objet. On peut (on doit) ajouter des méthodes appelées *constructeurs*. Le rôle d'un constructeur est de donner une valeur initiale (état initial) aux attributs.

Un constructeur est une *fonction membre* qui porte comme nom *le nom de la classe* et qui ne retourne rien. Un constructeur peut avoir zéro ou plusieurs arguments, éventuellement avec valeurs par défaut. Il peut y avoir plusieurs constructeurs dans la même classe (surcharge) dans la mesure où ils ont des signatures différentes. On peut donc compléter la classe en ajoutant des constructeurs.

```
-------------------------------------------------------------------------------
// point.h : définition de la classe
#ifndef __POINT__
#define __POINT__
#include<iostream>

class Point
{
public:
       // les constructeurs
       Point();
       Point(const unsigned x, const unsigned y);
```

```
        void SetXY(const unsigned x,const unsigned y);

         … le reste est identique …

private:
        unsigned _x;
        unsigned _y;
};

#endif
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
// point.cpp : définition des fonctions membres

#include "Point.h"
#include<math.h>

Point::Point()              // constructeur sans argument
{
        _x=0;
        _y=0;
}

Point::Point(const unsigned x, const unsigned y)   // constructeur à 2 arguments
{
        SetXY(x,y);    // pourquoi pas
}

… le reste est identique …
-------------------------------------------------------------------------------
```

Il faut bien remarquer, dans l'utilisation suivante, la syntaxe permettant d'utiliser les constructeurs. Les valeurs des arguments passés au constructeur sont données lors de la déclaration de l'objet.

```
-------------------------------------------------------------------------------
// utilisation de la classe Point munie de constructeurs
#include "Point.h"
using namespace std;

void main()
{
        // utilisation du constructeur Point::Point(const unsigned,const unsigned)
        Point p1(10,20);       // p1 est initialisé à l'aide du constructeur à 2 arguments
                               // l'état de p1 est (10,20)

        // utilisation du constructeur Point::Point()
        Point p2;              // p2 est initialisé avec le constructeur sans argument
                               // l'état de p2 est (0,0)

        p1.Display(cout);      // affiche (10,20)
        p2.Display(cout);      // affiche (0,0)
}
-------------------------------------------------------------------------------
```

De même, le C++ prévoit un mécanisme pour réaliser des opérations juste avant la mort d'un objet (par exemple, la désallocation de mémoire) . Il s'agit du *destructeur* de la classe. On reviendra sur ce point ultérieurement.

4.4. Différence entre sélecteurs/accesseurs et modificateurs.

Parmi les méthodes d'une classe, on distingue deux catégories : les méthodes qui modifient l'état de l'objet, méthodes appelées *modificateurs*, et les méthodes qui ne modifient pas l'état de l'objet mais qui y accèdent en lecture. Ces dernières sont appelées *accesseurs (celles qui utilisent l'état en lecture)* ou *sélecteurs (celles qui retournent l'état de l'objet)*. Il est important de déterminer dans quelle catégorie se situe une méthode car son prototype en dépend. Les sélecteurs/les accesseurs sont des méthodes déclarées *constantes* (il y a un `const` après la liste des arguments). Les modificateurs ont fréquemment un nom tel que **Set...()** et les sélecteurs **Get...().**

Pour la classe `Point`,

- les modificateurs sont `SetX(), SetY(), SetXY(), Translate()`
- les sélecteurs sont `GetX(), GetY(),`
- les accesseurs sont `Display(), Distance().`

Remarque : ni l'affichage ni le calcul de distance n'affecte l'état de l'objet sur lequel la méthode s'applique.

```
class Point
{
public:
       // modificateurs
       void SetX(const unsigned x);
       void SetY(const unsigned y);
       void SetXY(const unsigned x,const unsigned y);
       void Translate(const unsigned x,const unsigned y);

       // sélecteurs
       unsigned GetX() const;
       unsigned GetY() const;

       // accesseurs
       float Distance(const Point & p) const;
       void Display(std::ostream & flot) const;
};
```

4.5. Constructeur de copie, opérateur d'affectation

Parmi les constructeurs, le constructeur qui a un seul paramètre du même type que la classe est appelé *constructeur de copie*. Il sert à créer un nouvel objet ayant le même état qu'un objet de la même classe.

```
-------------------------------------------------------------------------------
// point.h : définition de la classe
#ifndef __POINT__
#define __POINT__
#include<iostream>

class Point
{
public:
       Point(const Point & p);       // prototype du constructeur de copie
       Point & operator=(const Point & p); // prototype de l'opérateur =
       ...
};
#endif
-------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
// point.cpp :
#include "Point.h"
Point::Point(const Point & p)
{
      // recopie de l'état de p dans l'objet courant
      _x=p._x;
      _y=p._y;
}

Point & Point::operator=(const Point & p)
{
      // operateur = pour l'affectation de l'état d'un Point
      _x=p._x;
      _y=p._y;
      return (*this);       // retourne l'objet courant.
}
...
-------------------------------------------------------------------------------
```

Dans le langage C++, l'affectation entre variables ou objets de même type est habituellement réalisée par l'opérateur =. Le langage C++ autorise la programmation des opérateurs agissant sur des types utilisateur (voir section suivante). On peut donc programmer le comportement de l'opérateur = portant sur des objets d'une classe. En C++, un opérateur est simplement traité comme une fonction particulière dont le nom de fonction comporte le mot réservé **`operator`**. L'opérateur d'affectation est donc simplement une méthode de la classe `Point`.

```
-------------------------------------------------------------------------------
// mise en évidence de l'utilisation du constructeur de copie et de l'opérateur =
#include "Point.h"
using namespace std;

void main()
{
      Point p1(10,20);
      Point p3;
      Point p2(p1);         // p2 est créé par copie de p1
                            // l'état de p2 est (10,20)

      p3=p1;                // équivaut à p3.operator=(p1);

      p3.Display(cout) ;    // affiche (10,20)
}
-------------------------------------------------------------------------------
```

4.5.1. Le pointeur this

Dans la définition de l'opérateur =, on voit l'utilisation d'un membre appelé `this`. Le membre `this` d'un objet est du type pointeur sur un objet de la classe. Pour la classe `Point`, `this` est donc un membre du type `Point *`. Ce membre contient l'adresse de l'objet courant. Autrement dit, tout objet peut connaître *sa propre adresse*. Logiquement, la notation `(*this)` représente *l'objet courant*, c'est-à-dire l'objet sur lequel la méthode a été appelée.

```
      Point & Point::operator=(const Point & p){
      _x=p._x;
      _y=p._y;
            return (*this);       // retourne l'objet courant.
      }
```

En résumé, l'opérateur = retourne l'objet sur lequel il s'applique. A quoi cela sert-il ? A garder l'homogénéité avec le C++ qui autorise les affectations successives telles que ci-dessous

```
void main(){
      Point p1(10,20), p2,p3;
      p3=p2=p1;      /* équivaut à p3.operator=(p2.operator=(p1)); */
}
```

Ci-dessus, `p2=p1` équivaut à `p2.operator=(p1)` et cette méthode retourne `p2`. C'est donc `p2` (après affectation) qui est affecté à `p3`.

5. Les opérateurs : fonctions globales ou méthodes d'une classe

Il est possible de surcharger la plupart des opérateurs. Cela signifie qu'on va pouvoir décrire quels traitements les opérateurs doivent réaliser. C'est notamment le cas pour l'opérateur = permettant l'affectation entre objets d'une même classe (voir section précédente). Cette surcharge n'est toutefois possible que sur les types créés par le programmeur : il n'est pas possible de redéfinir les opérateurs agissant sur les types élémentaires tels que `int`, `float`, etc.

5.1. La pluralité des opérateurs du C++

Selon le nombre d'opérandes dont l'opérateur a besoin, l'opérateur sera qualifié d'opérateur *unaire* (1 seul opérande) ou d'opérateur *binaire* (2 opérandes). Par exemple, l'opérateur « = » est un opérateur binaire, la syntaxe d'utilisation de cet opérateur étant `Op1 = Op2`. L' opérateur « ++ » en revanche est un opérateur unaire. En effet, il s'applique à un seul opérande : `Op1++`.

Remarque : il y a en fait deux opérateurs « ++ ». Celui qui pré-incrémente et qui s'utilise `++Op1` et celui qui post-incrémente et qui s'utilise `Op1++`. Idem pour les opérateurs de décrément «--» **.**

Enfin, de même qu'avec l'opérateur « ++ » où le même signe peut représenter deux opérateurs différents (la syntaxe d'utilisation permet au compilateur de les distinguer), certains opérateurs peuvent avoir une version unaire et une version binaire. C'est le cas par exemple de l'opérateur « - ».

```
#include<iostream>
      using namespace std;
void main()
{
      int a=4, b=5;
      cout << (a-b) << endl;       // opérateur - binaire
      cout << -a;                  // opérateur - unaire
}
```

La syntaxe **`a-b`** utilise l'opérateur – binaire (soustraction) tandis que **`–a`** utilise l'opérateur unaire (opposé).

5.2. Les opérateurs avec traitement par défaut

Lorsque l'on définit de nouveaux types, par exemple des types structurés, certains opérateurs réalisent un traitement par défaut. C'est le cas de l'opérateur « = » et de l'opérateur « & ».

Le traitement par défaut de l'opérateur = Lorsque l'on définit un nouveau type structuré (c'est le cas des classes), le traitement réalisé par défaut pour l'opérateur « = » est une copie membre à membre. Cette caractéristique s'appliquait déjà en langage C sur les types structurés.

Le traitement par défaut de l'opérateur `&` Lorsque l'on définit un nouveau type, l'opérateur `&` (opérateur unaire) retourne l'adresse de l'objet auquel il s'applique.

5.3. Les opérateurs que l'on peut surcharger en C++

Les différents opérateurs que l'on peut surcharger en C++ sont les suivants :

Unaire	`+ - ++ -- ! ~ * & new new[] delete (cast)`
Binaire	`* / % + - << >> < > <= >= == !=`
Binaire	`& | && ||`
Binaire	`+= -= *= /= %= &= ^= |= <<= >>= ,`

Les opérateurs membres d'une classe On verra ci-après que la définition des opérateurs passe simplement par l'écriture de fonctions ayant un nom particulier comportant le mot clé `operator` suivi du signe de l'opérateur. En outre, le programmeur aura le choix entre la possibilité de définir ces opérateurs comme fonctions membres d'une classe ou comme fonctions non membres. Néanmoins (le C++ est un langage truffé d'exceptions!) certains opérateurs ne peuvent être définis que s'ils appartiennent à une classe. Il s'agit alors nécessairement de méthodes d'une classe. C'est le cas pour l'opérateur = vu au chapitre précédent.

Doivent être impérativement définis comme opérateurs membres d'une classe les opérateurs suivants :

```
=       (opérateur d'affectation)
( )     (opérateur fonction)
[ ]     (opérateur d'indexation)
->
```

Remarque : rien n'oblige *a priori* le programmeur a faire de l'opérateur = un opérateur d'affectation, mais c'est tout de même recommandé. Sans quoi, la classe devient très difficile à comprendre par l'utilisateur !

5.4. Le mécanisme de définition des opérateurs en C++

Lorsqu'un opérateur est utilisé sur un type défini par le programmeur (une classe ou un type structuré), l'emploi de cet opérateur est équivalent à l'appel d'une fonction particulière qui peut être hors classe ou membre d'une classe (ou d'une structure). Si l'on définit une nouvelle classe `CLS`, et deux objets `x` et `y` de la classe `CLS`, alors la syntaxe (1) peut être équivalente à (2) **ou** (3) pour le compilateur C++ :

```
(1)     x+y;
(2)     operator+(x,y);       // opérateur non membre de la classe CLS
(3)     x.operator+(y);       // opérateur membre de la classe CLS
```

Autrement dit, les opérateurs sont vus comme des fonctions avec des identifiants particuliers : `operator` suivi du signe de l'opérateur. Dans le cas précédent, si l'on souhaite que l'opérateur + réalise un traitement particulier sur les variables de type `CLS` il suffit de définir une fonction (hors classe) appelée `operator+(CLS op1,CLS op2)` qui accepte deux arguments de type `CLS` (ou des références à `CLS`) *ou bien* d'ajouter une méthode `CLS::operator+(CLS op2)` à la classe `CLS`. On retrouve bien, dans les deux cas, les deux opérandes de l'opérateur +. Dans le premier cas, les deux opérandes sont les paramètres de la fonction. Dans le second cas, le premier opérande est l'objet sur lequel la fonction membre opérateur est appelée et le second le paramètre de la fonction membre.

5.5. Exemple : classe de nombres rationnels

Les opérateurs sont d'un usage particulièrement intéressant pour les objets mathématiques. On fournit ici une ébauche d'une classe de nombre rationnels mettant en évidence l'intérêt des opérateurs.

Les objets de la classe `Rationnel` sont simplement des fractions rationnelles où le numérateur et le dénominateur sont des entiers. On pourra ainsi manipuler des fractions du type 2/3, 27/11 sans erreurs d'arrondis, ou même des entiers (le dénominateur doit alors être égal à 1). Il faut noter qu'aucun objet de cette classe ne doit avoir de dénominateur nul (on teste donc cette éventualité au niveau du constructeur à

deux arguments). L'exemple donné ici illustre les deux façons de définir un opérateur (fonction membre ou hors classe).

```
-------------------------------------------------------------------------------
rationnel.h : fichier d'entête de la classe Rationnel
#ifndef __RATIONNEL__
#define __RATIONNEL__
#include<iostream>
class Rationnel
{
public:
	Rationnel(int num=0, int den=1);
	Rationnel(const Rationnel & r);

	int GetNum() const;
	int GetDen() const;

	// opérateurs membres de la classe
	Rationnel & operator=(const Rationnel & r);
	Rationnel operator+(const Rationnel & r) const;
private:
	int _num;
	int _den;
};
	// prototypes des opérateurs hors classe
	std::ostream & operator<<(std::ostream & flot, const Rationnel & r);
	Rationnel operator*(const Rationnel & r1, const Rationnel & r2);
#endif
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
// Rationnel.cpp : implémentation de la classe
#include "Rationnel.h"

// Constructeurs
Rationnel::Rationnel(int  num,int den){
	if(den==0) _den=1;    // éviter le dénominateur nul
	if(den<0){            // placer le signe au numérateur
		_den=-den;
		_num=-num;
	}
}

Rationnel::Rationnel(const Rationnel & r)  // constructeur de copie
{
	_den=r._den;
	_num=r._num;
}

// Sélecteurs
int Rationnel::GetNum() const {     return _num; }

int Rationnel::GetDen() const {     return _den; }

// opérateur pour l'affichage d'un rationnel (non membre)
std::ostream & operator<<(std::ostream & flot, const Rationnel & r)
{
	flot << r.GetNum() << "/" << r.GetDen();
	return flot;
}
```

```
// opérateur d'affectation (obligatoirement membre)
Rationnel & Rationnel::operator=(const Rationnel & r)
{
	_num=r._num;
	_den = r._den;
	return *this;
}

// opérateur pour la somme de nombres rationnels (membre)
Rationnel Rationnel::operator+(const Rationnel & r) const
{
	Rationnel resultat(_num*r._den+r._num*_den,_den*r._den);
	return resultat;
}

// opérateur pour le produit de nombres rationnels (hors classe )
Rationnel operator*(const Rationnel & r1,const Rationnel & r2) const
{
	// Attention, cet opérateur n'a pas accès aux membres privés !
	Rationnel resultat(r1.GetNum()*r2.GetNum(),r1.GetDen()*r2.GetDen());
	return resultat;
}
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
// utilisation de la classe Rationnel
#include "Rationnel.h"
using namespace std;
void main()
{
	Rationnel r1(2,3),r2(4,5);

	cout << " r1=" << r1;
	cout << " r2=" << r2 << endl;

	Rationnel r;

	r=r1+r2;	// equivaut à r.operator=(r1.operator+(r2))
	cout << "r1+r2=" << r << endl;

	r=r1*r2; // equivaut à r.opertator=(operator*(r1,r2))
	cout << "r1*r2=" << r;
}
-------------------------------------------------------------------------------
```

Dans l'exemple précédent il faut accorder une attention particulière aux prototypes. Il faut par exemple remarquer que la somme ne modifie ni l'objet courant (méthode constante) ni l'objet argument (référence constante). En outre, l'opérateur retourne un objet (différent de l'objet courant et de l'objet argument) de type rationnel correspondant au résultat de la somme.

6. L'utilisation de classes paramétrées en type (introduction à la bibliothèque STL)

6.1. Introduction

Le C++ introduit la notion de fonctions et de classes *paramétrées en type*. Cela signifie simplement qu'au sein d'une classe ou d'une fonction, certains types peuvent être passés en paramètre. Le mot clé *template* est introduit en C++ pour pouvoir paramétrer des types. Dans les manuels de C++, on trouvera

différentes appellations pour le paramétrage de type : on parlera de classes paramétrées, de patrons de classes ou de template-classes, voire de modèle de classes.

Une bibliothèque standard de classes paramétrées en type, libre d'utilisation, est fournie avec la plupart des compilateurs C++. Il s'agit de la bibliothèque **STL** (**Standard Template Library**). Elle regroupe différentes classes paramétrées et fonctions paramétrées. On y retrouve des classes *conteneur* paramétrées et des algorithmes paramétrés (recherche de valeurs dans un conteneur, tris de conteneur ...)

Notion de conteneur

Un conteneur est simplement un objet capable de stocker des données avec la possibilité de changer dynamiquement sa taille. Les conteneurs de la STL reprennent les structures de stockage de données les plus connues, à savoir

- le conteneur `vector` : tableau unidimensionnel
- le conteneur `list` : liste doublement chaînée
- le conteneur `stack` :pile (structure LIFO)
- le conteneur `queue` :file (structure FIFO)
- le conteneur `deque` : structure de données hybride
- le conteneur `set` : ensemble
- le conteneur `map` : tableau associatif

En plus de toutes ces structures de données, la bibliothèque STL fournit une classe `string` de gestion des chaînes de caractères. Le programmeur C++ tire bien des bénéfices à utiliser les objets `string` à la place des chaînes de caractères habituelles (la mémoire est allouée par les objets) et à utiliser des objets `vector` à la place des tableaux statiques ou mêmes dynamiques.

Les primitives d'accès, d'ajout, de suppression diffèrent bien évidemment entre ces conteneurs. Par exemple, l'accès à un élément d'un objet `vector` est un accès direct (indépendant de la taille du `vector`) alors que l'accès à un élément d'une liste chaînée est séquentiel : on doit parcourir la liste pour retrouver un élément. Par conséquent, certaines opérations sont plus efficaces sur certains conteneurs que sur d'autres. Ce sont ces caractéristiques qui guident le programmeur dans ses choix. Par exemple, on utilisera une liste chaînée plutôt qu'un tableau si l'on ajoute/retire souvent des éléments aux extrémités (en début ou en fin) puisque ces opérations sont plus efficaces sur une liste.

Tous les conteneurs STL sont paramétrés en type. Cela signifie qu'on peut choisir le type des données stockées dans le conteneur. On note `vector<T>` la classe obtenue à partir du conteneur `vector` pour le type `T`. Noter que `vector<int>` est une classe et `vector<float>` en est une autre.

```
-------------------------------------------------------------------------------
#include<iostream>          // entête pour les classe istream et ostream
#include<vector>            // entête pour le conteneur vector

using namespace std;        // la bibliothèque STL utilise le namespace std

void main(void)
{
       vector<int> tab1(4);         // objet de la classe vector<int>
       vector<float> tab2(2);       // objet de la classe vector<float>
       vector<vector<int> > tab3(2); //objet de la classe vector<vector<int> >

       tab1[1]=3;     // les éléments stockés dans tab1 sont des entiers
       tab2[0]=3.7;   // les éléments stockés dans tab2 sont des float
```

```
	tab3[0]=tab1;  // tab3 stocke des tableaux d'entiers (vector<int>)

	// affichage des éléments de tab1
	for(unsigned i=0;i<tab1.size();i++)
	{
	cout << tab1[i] << " ";
	}
}
-------------------------------------------------------------------------------
```

Ci-dessus `tab1` est un objet de la classe `vector<int>`. La classe `vector<int>` est une classe de tableaux dont les éléments stockés sont des entiers. L'objet `tab1` est donc un tableau d'entiers. La taille initiale du tableau est passée au constructeur de l'objet. L'objet `tab1` peut donc stocker 4 entiers, `tab2` peut stocker 2 float et `tab3` peut stocker 2 objets de la classe `vector<int>`. On peut donc copier `tab1` dans une case du tableau `tab3` (`tab3` est un tableau de tableaux).

Remarquer que l'accès à une case d'un objet `vector<T>` se fait simplement avec l'opérateur `[ ]`, comme sur un tableau classique en C. Mais la classe `vector<T>` dispose de bien d'autres méthodes utilisables pour changer la taille, insérer des éléments, supprimer des éléments. Par exemple, on voit ci-dessus l'utilisation de la méthode `vector<T>::size()` sur l'objet `tab1`.

L'objectif n'est pas de détailler ici toutes les possibilités d'utilisation des classes paramétrées de la bibliothèque STL, mais de présenter simplement quelques caractéristiques des classes `string`, `vector`, et `list`. Ces classes paramétrées vont fournir des briques intéressantes pour créer des classes plus complexes dans les chapitres suivants.

6.2. Namespace

Avant d'aller plus loin dans la présentation de la STL, il semble temps d'expliquer ce que représente un *namespace* (espace de noms). Différentes classes ou fonctions peuvent être regroupées dans un espace de noms particulier pour éviter certains conflits d'identifiant. Dès lors, pour utiliser une classe d'un espace de noms particulier, on doit faire précéder le nom de la classe du nom de l'espace de noms. On peut aussi préciser un `namespace` par défaut pour le programme via la directive `using namespace`.

Dans l'exemple qui suit, il n'y a pas de conflit de nom puisque les deux fonctions sont définies dans deux espaces de noms distincts (les namespaces `NA` et `NB`). On voit aussi que le flot `cout` dépend du namespace `std`.

```
-------------------------------------------------------------------------------
#include <iostream>

namespace NA{
	void MaFonction()
	{
		std::cout << "Fonction du namespace NA \n";
	}
}

namespace NB{
	void MaFonction()
	{
		std::cout << "Fonction du namespace NB \n";
	}
}
```

```
using namespace NA;  // utilise le namespace NA par défaut

void main(void)
{
       NA::MaFonction();     // fonction du namespace NA
       NB::MaFonction();     // fonction du namespace NB

       MaFonction();         // fonction du namespace par défaut
}
--------------------------------------------------------------------------------
```

Toutes les classes de la STL appartiennent au `namespace std`. Il convient donc de préciser ce `namespace` pour l'utilisation de ces classes. Ceci explique certaines parties de programme non compréhensibles jusque là.

Remarque : il est à noter que l'on ne peut pas utiliser la directive `using namespace` dans les fichiers d'entête lors des définitions de classes.

6.3. Classe `string`

La classe string n'est pas une classe paramétrée. Elle sert uniquement à gérer des chaînes de caractère de façon un peu plus souple qu'en langage C. Voici un premier exemple.

```
--------------------------------------------------------------------------------
#include<string>            // pour la classe string
#include<iostream>
using namespace std;

void main(void)
{
       string s1("Une chaine"),s2(" Toto");
       string s3(s1);
       string s4;

       s4=s1+s2;      // + : concaténation   = : affectation

       cout << s3 << endl;   // insertion possible dans cout
       cout << s3[0] << endl;// utilisable comme un tableau : s3[i] de type char

       if(s1==s3) cout << "Chaines identiques\n";

       if(s1<=s4) cout << "s1<=s4 selon l'ordre lexicographique\n";
}
--------------------------------------------------------------------------------
```

L'exemple suivant illustre les faits suivants : la classe contient

- un constructeur sans argument pour créer une chaîne vide
- un constructeur avec un argument de type char * pour assurer la compatibilité avec le C
- un constructeur de copie pour créer une chaîne identique à une chaîne existante
- des opérateurs de concaténation : + et +=
- un opérateur d'affectation : =
- un opérateur d'indexation [ ] de sorte qu'un objet string se comporte aussi comme un tableau
- les opérateurs de test : == !=
- les opérateurs de comparaison selon l'ordre lexicographique (l'ordre des mots dans un dictionnaire) : <= < >= >

La classe dispose également de différentes méthodes (présentation non exhaustive)

```
-------------------------------------------------------------------------------
void main(void)
{
	string s1("abcdefg");

	cout << "Longueur = " << s1.length() << endl;
	cout << "Capacite = " << s1.capacity() << endl;
	cout << s1 << endl;   // s1 = abcdefg

	s1.insert(1,"ABC");          // insère la chaîne "ABC" à l'indice 1
	cout << s1 << endl;          // s1 = aABCbcdefg

	s1.insert(2,4,'a');          // insère 4 fois la lettre 'a' à l'indice 2
	cout << s1 << endl;          // s1 = aAaaaaBCbcdefg

	s1.erase(0,1);               // supprime 1 caractère à l'indice 0
	cout << s1 << endl;          // s1 = AaaaaBCbcdefg

	s1.erase(2,5);               // supprime 5 caractères à l'indice 2
	cout << s1 << endl;          // s1 = Aabcdefg

	// possibilité d'obtenir un pointeur compatible const char *
	const char * ptr=s1.c_str();
}
-------------------------------------------------------------------------------
```

6.4. Conteneurs `vector<T>`

La classe paramétrée `vector<T>` permet de générer des objets tableaux pouvant stocker des objets de n'importe quel type. On rappelle que `vector<int>` est une classe (le paramètre de type vaut ici `int`) et `vector<float>` en est une autre. Par conséquent, ces types ne sont pas compatibles. Le premier exemple suivant met en évidence les principales fonctionnalités de ce patron de classes de tableaux.

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

void main(void)
{
	vector<int> t1(4,-2),t2(5);          // t1=-2,–2,–2,-2     t2=0,0,0,0,0
	vector<float> t3(2),t4(4,1.2);       // t3=0,0             t4=1.2,1.2,1.2

	t1[0]=5;       //t1=5,–2,–2,-2
	t4[2]=-2.3;    //t4=1.2,1.2,-2.3

	t2=t1; // possible car t1 et t2 de la même classe
	t3=t4; // possible car t3 et t4 de la même classe

	// t1=t4; impossible car t1 et t4 de types différents

	vector<int> t5(t1);   // t5=5,–2,–2,-2
	vector<float> t6(t3); // t6=1.2,1.2,-2.3

	//vector<float> t7(t1); impossible car t1 n'est pas de la classe vector<float>
}
-------------------------------------------------------------------------------
```

La classe dispose d'un constructeur à un argument (la taille initiale du vecteur), d'un constructeur à deux arguments (taille et valeurs initiales du vecteur). La classe dispose aussi d'un constructeur de copie. Attention, les copies ne sont possibles qu'entre objets de même classe ! La classe dispose d'un opérateur d'affectation. Là encore, l'affectation n'est possible qu'entre objets de même classe. Enfin, puisque cette classe implémente le fonctionnement d'un tableau, l'opérateur d'indexation [ ] donne accès aux éléments du tableau (comme pour un tableau classique).

Les itérateurs

Dans le but d'homogénéiser les interfaces des classes de la STL, les classes conteneur implémentent la notion *d'itérateur* qui généralise la notion de pointeur. Un itérateur est simplement un objet qui pointe un emplacement d'un conteneur (un peu comme un pointeur). En outre, les opérateurs `++` et `--` sont surchargés sur les itérateurs pour passer à l'emplacement suivant ou précédent du conteneur (qu'il s'agisse d'un tableau ou d'une liste chaînée). L'opérateur `*` permet l'indirection sur un itérateur (comme sur un pointeur). Grâce aux itérateurs, les algorithmes s'appliquant aux conteneurs sont moins dépendants de leur structure interne.

Les classes conteneurs contiennent des classes membres publiques pour instancier des itérateurs adaptés. Il y a deux classes d'itérateurs par conteneur : les itérateurs constants et les itérateurs non constants. Les premiers permettent de parcourir un conteneur sans pouvoir en modifier le contenu.

```
-------------------------------------------------------------------------------
void main(void)
{
	vector<int> t1(4);

	vector<int>::iterator it;	// it est un iterateur sur vector<int>

	for(it=t1.begin();it!=t1.end();it++) (*it)=2;

	for(unsigned i=0;i<t1.size();i++) t1[i]=2;
}
-------------------------------------------------------------------------------
```

Les deux boucles `for` précédentes réalisent le même traitement, l'une grâce aux itérateurs, l'autre grâce à l'opérateur `[]`. Les méthodes `begin()` et `end()` retournent des itérateurs. La première méthode retourne un itérateur sur le premier élément du conteneur. La seconde pointe juste après le dernier élément.

L'utilisation des conteneurs est donc un peu déroutante puisque des itérateurs sont utilisés comme arguments de certaines méthodes d'insertion ou de suppression. Il faut donc avoir une connaissance minimale sur leur utilisation pour pouvoir exploiter la bibliothèque STL. Néanmoins, l'utilisation des itérateurs est la même pour les autres conteneurs. En conséquence, dans la bibliothèque STL, l'utilisation d'une liste chaînée n'est pas plus compliquée que celle d'un vecteur. L'exemple suivant illustre les capacités d'insertion et de suppression dans un vecteur (voire dans un conteneur différent)

```
-------------------------------------------------------------------------------
void main(void)
{
	vector<int> t1(3,2); //t1=2,2,2
	cout << "Taille = "<< t1.size() << endl;	// Taille=3

	t1.push_back(3);	//t1=2,2,2,3

	t1.resize(6,-2);  //t1=2,2,2,3,-2,-2
```

```
	t1.insert(t1.begin(),2,-3); //t1=-3,-3,2,2,2,3,-2,-2

	t1.insert(t1.begin()+3,-1); //t1=-3,-3,2,-1,2,2,3,-2,-2

	t1.insert(&t1[0],-6); //t1=-6,-3,-3,2,-1,2,2,3,-2,-2

	t1.erase(t1.begin()+2,t1.begin()+4); //t1=-6,-3,-1,2,2,3,-2,-2

	t1.erase(t1.begin()+3); //t1=-6,-3,-1,2,3,-2,-2

	t1.pop_back(); //t1=-6,-3,-1,2,3,-2
}
--------------------------------------------------------------------------------
```

Ajout/retrait en fin = `push_back` et `pop_back`

Ajout/retrait en tête = `push_front` et `pop_front`

Premier/dernier élément : `front` et `back`

Changement de dimension : `resize`(nouvelle taille, valeur des cases complémentaires).

Insertion : plusieurs méthodes.

`insert`(itérateur, nombre d'ajouts, valeur insérée) : plusieurs cases insérées
`insert`(itérateur, valeur insérée) : une seule valeur insérée
Remarquer qu'un pointeur sur entier peut être converti en itérateur sur entier.

Suppression : plusieurs méthodes.

`erase`(itérateur debut,iterateur fin) : supprime les cases entre ces itérateurs
`erase`(itérateur) : supprime la case pointée par l'itérateur

6.5. Conteneurs `list<T>`

Les autres conteneurs ont de nombreux points communs avec ce qu'on a déjà vu. C'est pourquoi un exemple devrait suffire à comprendre l'utilisation des listes chaînées. Hormis le fait que l'opérateur `[ ]` n'est évidemment pas défini pour les listes chaînées.

```
--------------------------------------------------------------------------------
void main(void)
{

	list<int> l1;		//liste vide d'entiers
	list<float> l2,l4;	// liste vide de flottants

	l1.push_front(12);	// ajout en tête
	l1.push_back(13);	// ajout en fin
	l1.push_front(-3);	// ajout en fin
	l1.push_front(7);	// l1=7,-3,12,13

	l2.push_back(2.3);
	l2.push_back(2.7);
	l2.push_back(-1.2);	//l2=2.3,2.7,-1.2

	list<int> l3(l1);	//l3=7,-3,12,13
	cout << "Taille = "<< l3.size() << endl;
```

```
        l4=l2; //l4=2.3,2.7,-1.2

        l1.insert(l1.begin(),2,-3); //l1=-3,-3,7,-3,12,13

        cout << l1.front() << l1.back() << endl; // -3 13

        list<int>::iterator it=l1.begin();

        while(it!=l1.end()){
               if((*it)==-3)  // suppression des noeuds valant -3
               {
                      it=l1.erase(it);
               }
               else it++;
        }
        // affichage de la liste l1
        for(it=l1.begin();it!=l1.end();it++) cout << (*it) << " ";
}
-------------------------------------------------------------------------------
```

7. Réalisation de la composition en C++

Nous avons vu jusqu'ici comment utiliser les types primitifs pour écrire des classes et comment utiliser des classes standard (en particulier la classe `string` et les classes paramétrées `vector<T>` et `list<T>`). Très souvent, les classes vont également contenir comme attributs des objets d'une ou de plusieurs autres classes. Autrement dit, certains objets vont être *composés* d'objets d'autres classes. On peut par exemple, voir un objet de type voiture comme la composition (entre autres) d'un objet moteur, d'un objet carrosserie, de 4 objets roues ...

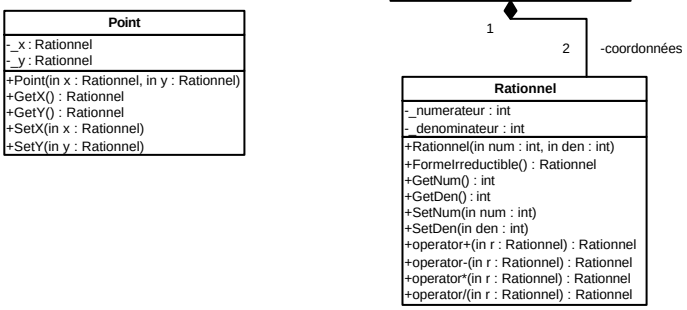
La réalisation la plus simple (mais pas nécessairement la seule en C++) de la composition est de faire apparaître les objets composants comme attributs de l'objet composite.

7.1. Représentation UML de la composition

Dans un premier temps, on va décrire le représentation UML de la composition. On peut décrire la composition de deux manières :

- soit les objets composants apparaissent comme attributs des objets composites
- soit une relation de composition (lien avec losange côté composite) lie les classes composant et composite.

Nous allons considérer deux exemples pour illustrer cette notion.

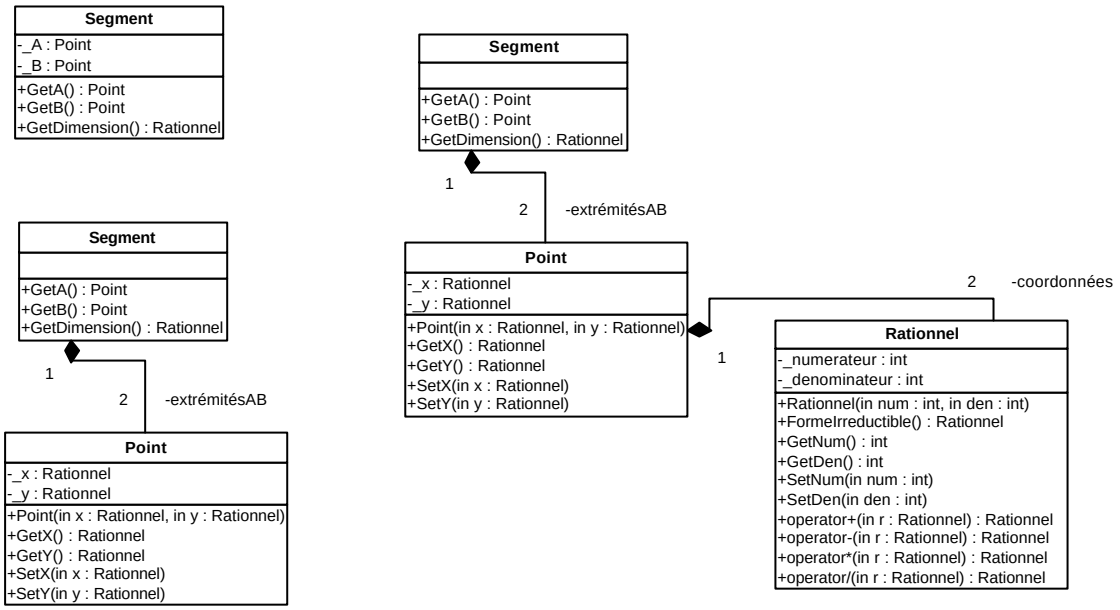


Classe `Point` : dans la première partie, un point du plan a été représenté par des coordonnées entières. De manière un peu différente, on peut aussi représenter des points à coordonnées rationnelles (voir la section sur les opérateurs).

Un point peut alors être vu comme la composition de deux coordonnées de type Rationnel (classe Rationnel).

En notation UML, les coordonnées sont soit représentées comme des attributs de type Rationnel dans la classe Point (représentation de gauche) soit par la relation de composition (représentation de droite). Dans le deuxième cas, la relation de composition est représentée par un losange côté composite.

Classe `Segment` : on peut décrire un segment par ses extrémités (objets de la classe `Point`). Là encore, on pourra avoir une représentation plus ou moins éclatée selon que l'on représente la composition comme attribut ou par une association particulière. Les trois représentations UML suivantes représentent la même classe `Segment` dont les extrémités sont des points à coordonnées rationnelles.



7.2. Objets composants comme attributs d'une classe

La réalisation la plus naturelle de la composition en C++ est de faire apparaître les objets composants comme des attributs de la classe composite. Sur le plan technique, seule l'écriture du constructeur introduit une notation particulière. Prenons l'exemple de la classe `Point` pour laquelle les coordonnées sont de type `Rationnel` (classe définie dans la section 5).

Seule la déclaration de la classe composant est utile pour l'écriture de la classe composite. On a seulement besoin de connaître les méthodes membre (notamment les constructeurs disponibles)

Rappelons tout d'abord le fichier d'entête de la classe composant puisque celui-ci contient toute l'information nécessaire à la réutilisation de la classe `Rationnel`.

```
-------------------------------------------------------------------------------
// rationnel.h : entête de la classe Rationnel (classe composant)

#ifndef _RATIONNEL__
#define _RATIONNEL__
class Rationnel
{
public:
       Rationnel(long num=0,long den=1);
       Rationnel(const Rationnel &);
       Rationnel & operator=(const Rationnel & );

       long GetDen() const;
       long GetNum() const;
```

```
        Rationnel FormeIrreductible();

        void Affiche(std::ostream & ) const;

        Rationnel operator+(const Rationnel & r) const;
        Rationnel operator*(const Rationnel & r) const;
        Rationnel operator-(const Rationnel & r) const;
        Rationnel operator/(const Rationnel & r) const;

private:
        long pgcd(long j,long k);
        long _num;
        long _den;
};
#endif
-------------------------------------------------------------------------------
```

Ensuite, la classe `Point` qui est une classe composite se définit de la manière suivante :

```
-------------------------------------------------------------------------------
//point.h : entête de la classe point (classe composite)

#ifndef _POINTRAT__
#define _POINTRAT__
#include "rationnel.h"      // inclusion de l'entête de la classe composant
class Point
{
public:
        Point();
        Point(const Rationnel & x, const Rationnel & y);
        Point(const Point & p);
        Rationnel GetX() const;
        Rationnel GetY() const;
private:
        Rationnel _x;         // objets membres de type Rationnel
        Rationnel _y;
};
#endif
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
//point.cpp (implémentation des méthodes de la classe Point)
#include "point.h"
Point::Point():_x(0,1),_y(0,1)
{
}

Point::Point(const Rationnel & x, const Rationnel & y):_x(x),_y(y)
{
}

Point::Point(const Point &p):_x(p._x),_y(p._y)
{
}

Rationnel Point::GetX() const {     return _x; }

Rationnel Point::GetY() const {     return _y; }
-------------------------------------------------------------------------------
```

Liste d'initialisation (en gras dans l'implémentation de la classe `Point`) : dans le fichier source de la classe `Point`, seule l'écriture des constructeurs introduit une nouvelle syntaxe. A la suite des arguments d'un constructeur, on peut mettre une *liste d'initialisation* des objets membres composants. Grâce à cette liste d'initialisation, le compilateur sait quel constructeur de la classe composant doit être invoqué pour initialiser l'objet membre.

Constructeur sans argument de la classe `Point` : initialisation de `_x` et `_y` comme les rationnels 0/1. Indique au compilateur que le constructeur à deux arguments de la classe `Rationnel` doit être utilisé.

Constructeur de copie de la classe `Point` : la liste d'initialisation indique que les attributs `_x` et `_y` sont initialisés grâce aux attributs `p._x` et `p._y` de l'objet `Point` passé en argument. Autrement dit, c'est le constructeur de copie de la classe `Rationnel` qui est invoqué.

7.3. Utilisation d'attributs de type tableau d'objets

Une autre réalisation possible de la composition consiste à mettre un attribut de type tableau d'objets. Pour la classe `Point`, on peut voir les coordonnées comme un tableau d'objets `Rationnel` à deux cases. Cette autre réalisation est mise en œuvre dans la classe `PointTab` ci-dessous.

```
-------------------------------------------------------------------------------
// pointtab.h
#ifndef _POINTTAB__
#define _POINTTAB__
#include "rationnel.h"
class PointTab
{
public:
      PointTab(const Rationnel & x, const Rationnel & y);
      Rationnel GetX() const;
      Rationnel GetY() const;
private:
      Rationnel _XY[2];     // tableau de deux objets Rationnel
};
#endif
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
//PointTab.cpp
#include "PointTab.h"

PointTab::PointTab(const Rationnel & x, const Rationnel & y)
{
      _XY[0]=x;
      _XY[1]=y;
}

Rationnel PointTab::GetX() const {  return _XY[0]; }
Rationnel PointTab::GetY() const {  return _XY[1];}
-------------------------------------------------------------------------------
```

Remarque : il est à noter qu'ici la liste d'initialisation ne permet pas d'initialiser le tableau d'objets. Aussi, il est nécessaire qu'un constructeur sans argument existe dans la classe `Rationnel`.

7.4. Utilisation d'un objet membre de type `vector<T>`

Une autre réalisation de la composition (où la multiplicité est supérieure à 1) consiste à utiliser objet membre de type `vector<Rationnel>`. Ceci implique assez peu de changements par rapport à la solution précédente. Seule l'initialisation de l'objet membre `vector<Rationnel>` doit faire l'objet d'une attention particulière.

```
-------------------------------------------------------------------------------
#ifndef _POINTTAB__
#define _POINTTAB__
#include "rationnel.h"
#include<vector>
class PointTab
{
public:
	PointTab(const Rationnel & x, const Rationnel & y);
	Rationnel GetX() const;
	Rationnel GetY() const;
private:
	std::vector<Rationnel> _XY;
};
#endif
-------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
//PointTab.cpp
#include "PointTab.h"
// Attention ! Utiliser la liste d'initialisation pour initialiser le vecteur à la taille
2
PointTab::PointTab(const Rationnel & x, const Rationnel & y):_XY(2)
{
	_XY[0]=x;
	_XY[1]=y;
}

Rationnel PointTab::GetX() const {  return _XY[0]; }

Rationnel PointTab::GetY() const {  return _XY[1];}
-------------------------------------------------------------------------------
```

Remarque : une autre réalisation possible de la composition est proposée dans la section traitant des classes avec données en profondeur.

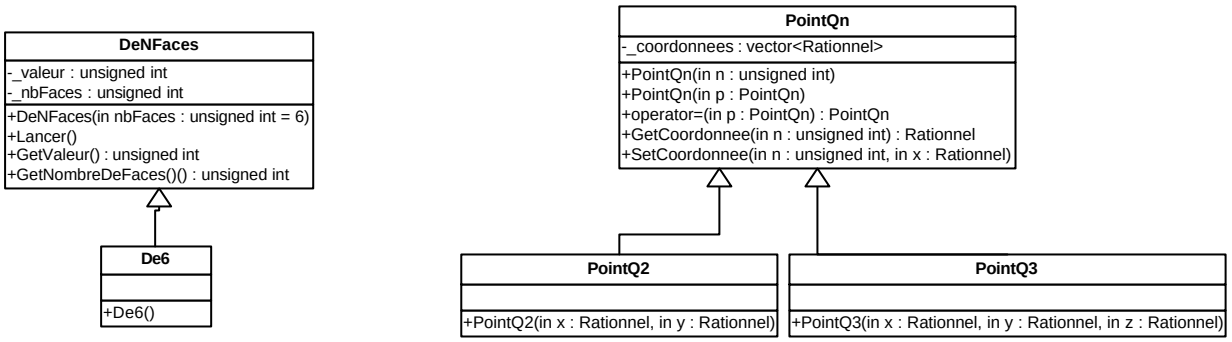
8. Réalisation de la spécialisation en C++

Le relation de spécialisation (ou de généralisation) qui existe entre des classes est réalisable en C++. La possibilité d'écrire une sous-classe (ou classe spécialisée) d'une classe existante est prévue par ce langage, de même que pour tous les langages objet.

8.1. Représentation de la spécialisation en UML

Le langage UML propose une notation (une flèche pointant la super-classe) pour décrire la relation de spécialisation entre classes. La classe spécialisée (ou sous-classe d'un point de vue ensembliste) possède les attributs et le comportement de la super-classe. On dit aussi que la sous-classe hérite de la super-classe. En plus du comportement hérité, la sous-classe peut contenir des attributs et des méthodes spécifiques. La spécialisation est représentée par une flèche en langage UML.

Exemple : une classe de dés à 6 faces peut être vue comme une spécialisation d'une classe de dés à n faces : il suffit de fixer le nombre de faces.



Exemple : on peut aussi voir un point du plan QxQ (l'ensemble des couples de nombres rationnels), classe notée `PointQ2`, comme la spécialisation d'un point à n coordonnées (classe `PointQn`)

On retrouve dans la bibliographie C++ un certain vocabulaire associé à cette notion de spécialisation/généralisation. Au sujet de la classification ci-dessus, on dira :

- la classe `De6` est une spécialisation de la classe `DeNFaces`
- la classe `De6` dérive de la classe `DeNFaces`
- la sous-classe `De6` hérite des fonctionnalités de la super-classe `DeNFaces`. En effet, un objet d'une sous-classe dispose également de l'interface de la super-classe.
- la classe `DeNFaces` est la super-classe de la classe `De6`
- la classe `DeNFaces` est la classe de base de la classe `De6`.

8.2. Exemple de réalisation de la spécialisation en C++

Nous donnons l'exemple de la classe dérivée `De6`. Nous rappelons d'abord le fichier header (fichier d'entête) de la classe `DeNFaces` car il est nécessaire de connaître l'interface de la super-classe pour écrire la classe dérivée. Ensuite nous donnons la définition de la classe dérivée (la sous-classe) ainsi que l'utilisation de cette classe. Pour cet exemple, la classe dérivée n'a pas d'attribut supplémentaire. La classe dérivée se contente de contraindre la valeur du nombre de faces. Un dé à 6 faces est un dé à N faces où _nbFaces vaut toujours 6.

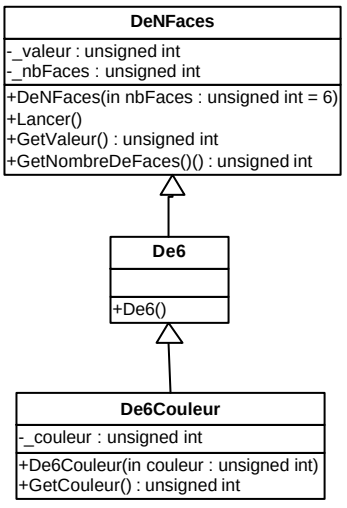
Pour la classe spécialisée, seul le constructeur sans argument est défini pour initialiser correctement le nombre de faces. Ensuite, toutes les méthodes de la classe `DeNFaces` peuvent être utilisées sans problème sur la classe spécialisée `De6`.

```
-------------------------------------------------------------------------------
// DeNFaces.h : entête d'une classe de dés à N faces
#ifndef __CLS_DENFACES__
#define __CLS_DENFACES__
class DeNFaces
{
public:
      DeNFaces(unsigned nbFaces=6);
      unsigned GetNombreDeFaces() const;
      void Lancer();
      unsigned GetValeur() const;
private:
      unsigned _valeur;           // valeur du dé à un instant
      unsigned _nbFaces;          // nombre de faces du dé
};
#endif
-------------------------------------------------------------------------------
```

L'implémentation de la classe dérivée est ici très simple puisque seul un constructeur est défini. Encore une fois, la *liste d'initialisation* (revoir la section précédente) est utilisée au niveau des constructeurs.

```
-------------------------------------------------------------------------------
// De6.h : fichier d'entête de la classe De6 (dérivée de DeNFaces)
#ifndef __CLS_DE6__
#define __CLS_DE6__
#include "DeNFaces.h"

class De6:public DeNFaces           // la classe De6 dérive de DeNFaces
{
public:
      De6();        // constructeur de la classe dérivée
};
#endif
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
// De6.cpp :implémentation de la classe spécialisée
#include "De6.h"
De6::De6():DeNFaces(6){ }
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
// Utilisation.cpp : utilisation de la classe sépcialisée
#include "De6.h"
#include<iostream>
using namespace std;
void main(void)
{
      DeNFaces de1(9);      // dé à 9 faces
      De6 de2;              // dé à 6 faces
      cout << de1.GetNombreDeFaces() << "\n";
      cout << de2.GetNombreDeFaces() << "\n";
      de1.Lancer();
      de2.Lancer();  // appel de méthode héritée
      cout << d1.GetValeur() << endl;
      cout << d2.GetValeur() << endl; // appel de méthode héritée
}
-------------------------------------------------------------------------------
```

Notons que l'on peut encore poursuivre la hiérarchie de classes d'objets dés. On peut par exemple ajouter un attribut de couleur .




```
-------------------------------------------------------------------------------
#ifndef __CLS_DE6COUL__
#define __CLS_DE6COUL__
#include "De6.h"
class De6Couleur:public De6
{
public:
      De6Couleur(unsigned couleur);
      unsigned GetCouleur() const;
private:
      unsigned _couleur;
};
#endif
---------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
// De6Couleur.cpp : implémentation de la classe de De6Couleur
#include "De6Couleur.h"
De6Couleur::De6Couleur(unsigned c):De6(),_couleur(c)
{
}
unsigned De6Couleur::GetCouleur() const
{
	return _couleur;
}
-------------------------------------------------------------------------------
```

8.3. Les conversions de type entre sous-classe et super-classe.

Une sous-classe représente un sous-ensemble d'objets de la classe plus générale. Si l'on se réfère à la hiérarchie de classes de dés précédente, on comprend facilement qu'un dé à 6 faces est avant tout un dé à N faces. Aussi, le type `De6` peut être converti en type `DeNFaces`.

En C++, les conversions de type entre sous-classe et super classe sont légales. Par exemple, on peut convertir un objet `De6` en objet `DeNFaces`. On peut de la même manière convertir un objet de la classe `De6Couleur` en un objet de la classe `DeNFaces`. Néanmoins, ces conversions sont dégradantes (perte de données), comme lorsque l'on convertit un `float` en `int`. Naturellement, lorsque l'on convertit un dé coloré en dé (sans couleur) on perd l'information de couleur. En C++, les conversions sous-classe vers super-classe sont possibles, mais elles perdent la partie dite « incrémentale » c'est-à-dire la partie spécifique d'une classe dérivée. L'exemple suivant présente quelques transtypages possibles.

```
-------------------------------------------------------------------------------
#include "De6Couleur.h"
#define rouge 5
void main(void)
{
	De6Couleur de6r(rouge);
	De6 de6;
	DeNFaces denf(9);

	((DeNFaces)de6r).GetValeur();		// cast De6Couleur -> DeNFaces

	((DeNFaces)de6).GetValeur();  // cast De6 -> DeNFaces
}
-------------------------------------------------------------------------------
```

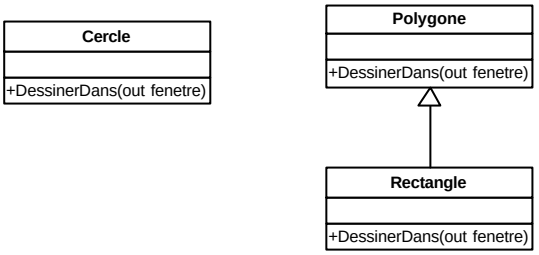
Conversion sous-classe * -> super-classe * : les conversions de pointeurs entre sous-classe et super-classe sont également légales en C++.

```
-------------------------------------------------------------------------------
#include "De6Couleur.h"
#define rouge 5
void main(void)
{
	De6Couleur de6r(rouge);
	De6 *ptrDe6;
	DeNFaces *ptrDeNF;

	ptrDe6=&de6r;		// De6Couleur * -> De6 *
	ptrDe6->Lancer();
```

```
        ptrDeNF=ptrDe6;       // De6 * -> DeNFaces *
        ptrDeNF->Lancer();
}
-------------------------------------------------------------------------------
```

8.4. Le polymorphisme et les méthodes virtuelles

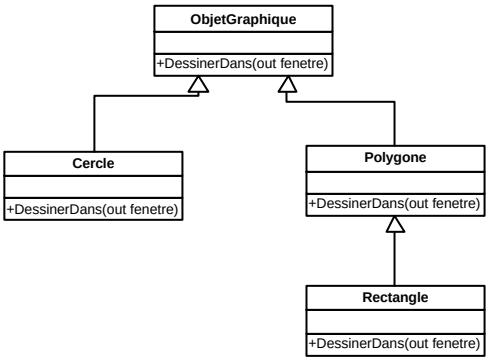
Le polymorphisme représente le fait qu'une fonction ayant le même nom puisse être appelée sur des objets de classes différentes. Par exemple, le fait qu'on puisse mettre une méthode `DessinerDans(fenetre)` dans différentes classes peut être vu comme du polymorphisme. En effet, pour la classification faite ci-dessous, les classes représentent des objets graphiques destinés à être dessinés dans des fenêtre sous windows. La fonction de dessin est donc présente dans les différentes classes.



En C++, le polymorphisme représente aussi la possibilité pour des objets d'une descendance à répondre différemment lors de l'appel d'une méthode de même nom. Le polymorphisme est donc lié dans ce cas aux hiérarchies de classes.

Si l'on reprend la classification précédente et que l'on ajoute une super-classe `ObjetGraphique`, on voit que tous les objets graphiques ont besoin d'une méthode permettant de dessiner l'objet dans une fenêtre. De plus, la méthode de dessin doit agir différemment sur les différents objets graphiques : un rectangle ne se dessine pas de la même manière qu'un cercle. Si la classe de base de la descendance contient une méthode `DessinerDans()`, en raison des compatibilités de type entre sous-classe et super-classe, on peut donc exécuter le programme suivant.

```
-------------------------------------------------------------------------------
#include "Rectangle.h"
#include "Cercle.h"
#include<iostream>
using namespace std;

void main(void)
{
Fenetre f; // objet fenêtre graphique
ObjetGraphique * tab[3];

tab[0]=new Cercle;
tab[1]=new Rectangle;
tab[2]=new Polygone

tab[0]->DessinerDans(f);    // méthode de la classe ObjetGraphique invoquée
tab[1]->DessinerDans(f);    // car la méthode n'est pas polymorphe par défaut
tab[2]->DessinerDans(f);

for(int i=0;i<3;i++) delete tab[i];
}
-------------------------------------------------------------------------------
```



Dans l'exemple précédent la méthode `DessinerDans()` peut être invoquée sur tous les objets. Mais, si la méthode `DessinerDans()` n'est pas déclarée **`virtual`** dans la classe de base `ObjetGraphique`, c'est nécessairement la méthode de la classe `ObjetGraphique` qui va être invoquée, même si l'objet pointé est de la classe `Rectangle` ou `Cercle`. Par défaut en C++, un **lien statique** relie les méthodes aux classes. Autrement dit, par défaut, les méthodes ne sont pas polymorphes. Il faut préciser quand on souhaite que le polymorphisme s'applique.

Remarque : en Java, le polymorphisme s'applique automatiquement. Le C++ laisse la possibilité de mettre en place ou non le polymorphisme (via les méthodes virtuelles) pour des raisons de performance. Car la définition de méthodes virtuelles (voir ci-dessous) implémente une table d'indirection supplémentaire qui augmente la taille du code et ralentit l'exécution. Par conséquent en C++, selon le besoin, le programmeur peut privilégier la vitesse quand le polymorphisme n'est pas utile.

Les méthodes virtuelles. Si l'on définit la méthode `DessinerDans()` comme étant virtuelle **(mot clé `virtual`)** dans la classe de base **`ObjetGraphique`**, le programme précédent va bien invoquer les méthodes `DessinerDans()` de chacun des objets pointés. La méthode adaptée à l'objet est retrouvée *dynamiquement*. Le mot clé `virtual` indique au compilateur de mettre en place une *ligature dynamique* de méthodes. Le compilateur crée alors une table d'indirection (ceci est transparent pour le programmeur) permettant de retrouver la bonne méthode à l'exécution. Le choix de la méthode invoquée a alors lieu à l'exécution et non à la compilation.

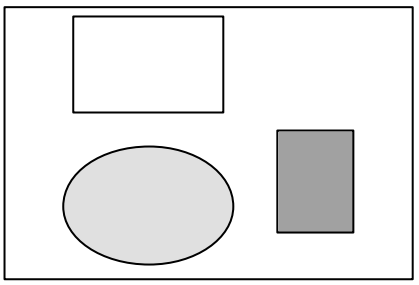
```
-------------------------------------------------------------------------------
#ifndef __OBJETGRAPHIQUE__
#define __OBJETGRAPHIQUE__
#include "Fenetre.h"

class ObjetGraphique
{
public:
      ObjetGraphique();
      virtual void DessinerDans(Fenetre &) const;
};
#endif
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
#ifndef __RECTANGLE__
#define __RECTANGLE__
#include "ObjetGraphique.h"

class Rectangle: public ObjetGraphique
{
public:
      Rectangle();
      virtual void DessinerDans(Fenetre &) const;
};
#endif
-------------------------------------------------------------------------------
```

Remarque : il faut noter que la ligature dynamique (obtenue par le mot clé `virtual`) ne concerne que la méthode précisée, et non toutes les méthodes de la classe.

En C++, quand on parle de polymorphisme, on pense en priorité à ce lien dynamique mis en place en utilisant le mot clé `virtual`.

8.5. Les classes abstraites

Grâce au polymorphisme on peut écrire du code générique et faire des structures de données hétérogènes. Pour comprendre ces idées, le plus simple est de considérer de nouveau la classification d'objets graphiques faite précédemment. On souhaite faire un petit programme de dessin où l'utilisateur peut dessiner des formes géométriques simples (ellipses, rectangles....), les sélectionner, les supprimer, les déplacer ... Pour cela, les formes géométriques doivent être mémorisées dans une structure de données. En outre, toutes les formes géométriques devront pouvoir réagir à des mêmes demandes (mêmes méthodes).



L'interface de la classe de base `ObjetGraphique` va contenir tout ce que les différentes formes géométriques seront capable d'exécuter. Mais la notion d'objet graphique reste abstraite. Que signifie « dessiner un objet graphique » ? ou même « déplacer un objet graphique » ?

En fait, la classe `ObjetGraphique` va être une classe dite *abstraite*. Elle contiendra des fonctions virtuelles pures, c'est-à-dire des fonctions membres qui ne seront pas définies (en outre on ne saurait pas quoi y mettre) mais pour lesquelles le typage dynamique s'appliquera. Les fonctions virtuelles pures n'ont pas de code et ont un prototype qui se termine par =0. Une classe qui contient au moins une fonction virtuelle pure est dite abstraite. On ne peut alors pas créer d'objet d'une telle classe (mais on pourra créer des objets d'une classe dérivée).

```
-------------------------------------------------------------------------------
#ifndef __OBJETGRAPHIQUE__
#define __OBJETGRAPHIQUE__
#include "Fenetre.h"
#include "Point.h"
class ObjetGraphique        // classe abstraite
{
public:
      ObjetGraphique();
      virtual ~ObjetGraphique();   // destructeur virtuel

      // fonctions virtuelles pures
      virtual void DessinerDans(Fenetre &) const =0;
      virtual void Deplacer(int DeltaX, int DeltaY) =0 ;
      virtual void Contient(const Point &) const =0 ;
      …
};
#endif
-------------------------------------------------------------------------------
```

On ne peut pas créer d'objet de la classe `ObjetGraphique`, mais en revanche, on peut faire un conteneur d'objets graphiques. En effet, on peut faire par exemple un tableau de pointeurs dans lequel on peut placer les adresses des différentes formes gérées par le programme. Le programme ci-dessous gère une

telle structure de données contenant des objets hétérogènes. Ils sont en revanche tous descendants de la classe `ObjetGraphique`.

```
void main()
{
	vector<ObjetGraphique *> FormesGraphiques;
	Fenetre F ;

	FormesGraphiques.push_back(new Rectangle(10,10,40,55));
	FormesGraphiques.push_back(new Ellipse(60,60,140,155));
	FormesGraphiques.push_back(new Rectangle(30,30,40,40));

	for(int i=0;i<FormesGraphiques.size();i++) FormesGraphiques[i]->DessinerDans(F);
	for(int i=0;i<FormesGraphiques.size();i++) delete FormesGraphiques[i];
}
```

De plus, on voit qu'il est très simple de redessiner l'ensemble des objets (code en gras), puisque on envoie le même message à chaque objet, ce message étant interprété différemment par chacun des objets (grâce au typage dynamique). En résumé, une classe abstraite décrit ce qu'on attend d'une classe (comportement) sans savoir comment cela va être réalisé. En fait le comportement final est programmé dans la ou les classes descendant de la classe abstraite.

9. Visibilité des membres d'une classe

Jusqu'ici, nous avons utilisé uniquement des membres (attributs ou méthodes) de type `public` ou `private`. Ces mots-clé règlent la visibilité des membres. Les membres `public` d'une classe sont accessibles à la fois aux objets de la classe ainsi qu'à l'utilisateur de la classe (comme dans les types structurés du C). Toutes les méthodes de l'interface d'une classe (c'est-à-dire le comportement d'une classe) sont à accès `public`. En revanche, l'encapsulation des données d'une classe est possible grâce au mot-clé `private`. Les membres `private` sont accessibles aux objets de la classe mais pas de l'utilisateur de la classe. L'utilisateur ne peut pas agir directement sur les données membres d'un objet.

Dans cette section, on présente tout d'abord comment des fonctions non membres d'une classe, ou comment d'autres classes, peuvent avoir également accès à la partie encapsulée d'une classe donnée. Il s'agit des fonctions ou classes *amies* (mot clé `friend`).

Ensuite, on présente un nouveau droit d'accès : un membre *protégé* (mot-clé `protected`) est comme un membre privé (non accessible par l'utilisateur) mais reste accessible pour une classe dérivée. C'est donc dans le cadre de la dérivation que ce mot clé a un usage. Enfin, on révisera les problèmes de visibilité dans les hiérarchies de classes.

9.1. Mot-clé `friend` : fonctions ou classes amies

On peut déclarer, dans une classe `CLS`, le fait que d'autres fonctions ou d'autres classes aient accès à la partie encapsulée de la classe `CLS`. Cette fonctionnalité est particulièrement intéressante lorsque l'on veut définir des opérateurs comme fonctions non membres de la classe.

On illustre ceci en reprenant la classe `Rationnel` vue dans la section sur les opérateurs (section 5) et en définissant les opérateurs `+` et `*` comme des fonctions non membres de la classe `Rationnel` mais amis de cette classe. Les membres `_den` et `_num` deviennent accessibles à ces fonctions non membres.

```
class Rationnel
{
	// fonctions non membres mais amies de la classe
	friend Rationnel operator+(const Rationnel & r1,const Rationnel & r2);
	friend Rationnel operator*(const Rationnel & r1,const Rationnel & r2);
	...
};
```

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

Rationnel operator+(const Rationnel & r1,const Rationnel & r2){
       Rationnel local(r1._num*r2._den+r2._num*r1._den,r1._den*r2._den);
       return local;
}
Rationnel operator*(const Rationnel & r1,const Rationnel & r2){
       Rationnel local(r1._num*r2._num,r1._den*r2._den);
       return local;
}
```

On peut également déclarer une classe amie. Dans l'exemple suivant, la classe `B` est déclarée amie de la classe `A`. Cela signifie que toutes les méthodes de la classe `B` peuvent avoir accès à la partie privée d'un objet de la classe `A`.

```
#include "B.h"
class A
{
public:
       friend B;      // la classe B est amie de la classe A
       …
};
```

Remarque : la déclaration d'amitié permet de contourner l'encapsulation des données d'une classe. Il ne faut donc pas abuser de ce droit. Il est préférable de chercher à éviter autant que possible d'utiliser l'amitié (on peut souvent procéder différemment en ajoutant des accesseurs notamment).

9.2. Membres protégés (mot-clé `protected`) et visibilité des membres

Un membre (donnée ou méthode) déclaré `protected` est accessible par les objets de la classe mais pas par l'utilisateur de la classe (comme pour un membre privé). En revanche, à la différence d'un membre `private`, un membre `protected` est accessible par un objet d'une classe dérivée. On déclare donc `protected` un membre que l'on souhaite rendre accessible a une classe dérivée. En résumé, pour une classe (classe `Base`) et une classe dérivée (classe `Derivee`) donnée on a :

```
-------------------------------------
#ifndef ___CLS_BASE__
#define ___CLS_BASE__
class Base
{
public:
       Base();
       MethodeB();

protected:
       bool _indPProtegeB;
       void FonctionProtegeeB();

private:
       int _dPriveeB;
       void FPriveeB();
};
#endif
-------------------------------------
```

```
-------------------------------------
#ifndef ___CLS_DERIVEE__
#define ___CLS_DERIVEE__

#include "Base.h"
class Derivee : public Base
{
public:
       Derivee();
       MethodeD();

protected:
       bool _indProtegeD;
       void FonctionProtegeeD();

private:
       int _dPriveeD;
       void FPriveeD();
};
#endif
-------------------------------------
```

Tous les membres privés et protégés de la classe `Base` sont accessibles dans les méthodes (privées ou protégées) de la classe `Base`. Par contre un utilisateur de la classe `Base` n'a accès qu'à la méthode publique de cette classe.

De même, tous les membres privés et protégés de la classe `Derivee` sont accessibles dans les méthodes (privées ou protégées) de la classe `Derivee`. En outre, du fait de la dérivation, les méthodes publiques de la classe `Base` sont utilisables sur les objets de la classe `Derivee`.

```
void main()
{
      Derivee D ;
      D.MethodeD() ;
      D.MethodeB() ;        // méthode héritée
}
```

Dans une méthode privée ou protégée de le classe `Derivee`, on peut accéder à la partie publique de la classe de base (normal) et à la partie *protégée* de la classe de base.

```
void Derivee::FPriveeD()    // implémentation de la classe Derivee
{
      MethodeB();           // possible (car publique)
      MethodeD();           // possible (car publique)

      _indPProtegeB=true;   // possible (car membre protégé de Base)
      // _dPriveeB=3;       // non possible (car membre privé de Base)

      FonctionProtegeeB();  // possible (car fonction protégée de Base)
      //FPriveeB();         // non possible (car fonction privée de Base)
}
```

Trois formes de dérivation sont possibles : la dérivation publique, privée ou protégée.

Tout ce qui vient d'être exposé ne tient que pour la dérivation publique (`class Derivee : public Base`). C'est le mode de dérivation le plus courant. Il existe aussi une dérivation privée (`class Derivee : private Base`) et protégée (`class Derivee : protected Base`). La visibilité des membres de la classe de base dépend du mode de dérivation. On résume ci-dessous la visibilité des membres selon le mode de dérivation. Par la suite, FMA signifie Fonctions Membres et Amies

La dérivation publique (`class Derivee : public Base`)

Statut dans la classe de base	Accès aux FMA de la classe dérivée	Accès à un utilisateur de la classe dérivée	Nouveau statut dans la classe dérivée[2]
Public	Oui	Oui	Public
Protégé	Oui	Non	Protégé
Privé	Non	Non	Privé

La dérivation privée (`class Derivee : private Base`)

Dans ce mode de dérivation, pour un utilisateur de la classe dérivée, tout ce qui est hérité de la classe de base est encapsulé par la dérivation. C'est un peu comme si l'on définissait un membre privé de la classe de base dans la classe dérivée. Autrement dit, on n'hérite pas vraiment des méthodes.

[2] en cas de nouvelle dérivation

La dérivation protégée (`class Derivee : protected Base`)

Dans ce mode de dérivation, les membres publics de la classe de base seront considérés comme protégés lors des dérivations ultérieures.

Tableau de synthèse

Classe de base			**Dérivée publique**		**Dérivée protégée**		**Dérivée privée**	
Statut initial	Accès FMA	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur
Public	Oui	Oui	Public	Oui	Protégé	Non	Privé	Non
Protégé	Oui	Non	Protégé	Non	Protégé	Non	Privé	Non
Privé	Oui	Non	Privé	Non	Privé	Non	Privé	Non

Ce tableau est difficile à assimiler. Seule la pratique de la programmation permet de retenir ces aspects du langage. Il faut dans un premier temps se concentrer sur la dérivation publique qui est le mode de dérivation le plus couramment employé.

9.3. Navigation dans une hiérarchie de classes, phénomènes de masquage

On s'intéresse ici à ce que va pouvoir exploiter un utilisateur dans le cas d'une hiérarchie de classes. Considérons les classes partiellement décrites ci-dessous.

```
class Base
{
public:
      Base();
      void Affiche() const;
      void AfficheB() const;
      …
};
```

```
class Derivee:public Base
{
public:
      Derivee();
      void Affiche() const;
      void AfficheD() const;
      …
};
```

La classe dérivée comporte, comme la classe de base, une méthode `Affiche()`. Sur un objet `Derivee`, la méthode `Affiche()` appelée est nécessairement celle de la classe dérivée. On dit qu'elle masque celle de la classe de base. Mais, cette dernière demeure utilisable en précisant à l'appel le nom de la classe propriétaire. D'ailleurs on peut toujours préciser le nom de la classe lors d'un appel de méthode. Par exemple, l'appel ci-dessous est tout à fait légal (mais pas très concis)

```
      void main()
      {
      Base B;
      B.Base::Affiche();    // équivaut à B.Affiche();
      }
```

```
-----------------------------------------------------------------------------------------
#include "derivee.h"
void main()
{
      Derivee D;
      Base B;

      B.Affiche();
      B.AfficheB();

      D.AfficheD();
      D.Affiche();          // Derivee::Affiche() masque ici Base::Affiche()
```

```
        D.Base::Affiche();    // appel explicite de la méthode de la classe de base
        D.Derivee::Affiche() ;       // équivalent à D.Affiche();
        B.Base::AfficheB();          // équivalent à B.AfficheB() ;

        // B.Derivee::AfficheD(); n'a en revanche pas de sens
    }
    ------------------------------------------------------------------------------------
```

10. Classes avec données en profondeur

10.1. Les opérateurs de gestion de mémoire (new et delete)

En C, on demande de la mémoire avec la fonction `malloc()` et on la libère avec la fonction `free()`. En C++, l'allocation dynamique de mémoire s'effectue avec les opérateurs `new` et `delete`.

10.1.1. Syntaxe

La syntaxe d'utilisation de l'opérateur `new` est la suivante :

```
T * ptr;
ptr = new T;         // allocation d'un objet (ou une variable) de type T
ptr = new T[n];      // allocation d'un tableau de n objets de type T
```

où `T` est un identificateur de type quelconque (type primitif, type structuré ou classe), `n` est une expression entière quelconque.

La première syntaxe réserve de la mémoire pour stocker 1 élément ayant le type `T`, alors que la seconde réserve de la mémoire pour stocker n éléments de type `T`. Dans les deux cas, le résultat de l'opération est une adresse. Soit l'adresse de l'espace mémoire réservé sous la forme d'un pointeur de type `(T *)`, soit le pointeur `NULL` si l'espace demandé n'a pu être obtenu.

L'utilisation de l'opérateur `delete` est la suivante :

```
delete ptr;
delete [] ptr;
```

où `ptr` est un pointeur ayant obtenu sa valeur actuelle par l'opérateur `new`.

La première syntaxe libère l'espace mémoire d'un objet pointé par `ptr`. La seconde syntaxe sert à libérer l'espace occupé par un tableau d'objets. Cette seconde syntaxe assure que tous les destructeurs des objets du tableau sont appelés avant que le tableau soit libéré. Ceci est important si les objets du tableau ont des données en profondeur (voir la suite de ce chapitre). Il est à noter qu'il n'est pas nécessaire de rappeler la taille du tableau alloué lors de la libération de la mémoire.

Remarque : les objets string de la STL ont des données en profondeur. Le code suivant conduit à des fuites mémoire

```
string * ptr;
ptr = new string[6];
delete ptr;
```

Remarque : le fonctionnement de l'opérateur `delete` est indéterminé si `ptr` pointe sur une zone qui n'a pas été allouée dynamiquement par l'opérateur `new`, ou si la zone mémoire a déjà été libérée. Il est déconseillé de mélanger l'utilisation des opérateurs `new/delete` avec celle des fonctions `malloc()/free()`.

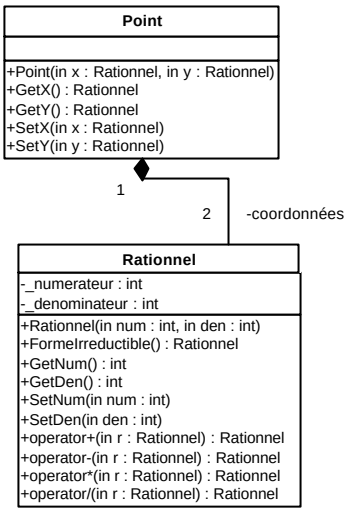
On peut préciser l'utilisation d'un constructeur particulier lors de l'allocation d'un objet (mais pas pour les tableaux alloués dynamiquement). Ci dessous, l'opérateur `new` alloue la mémoire pour stocker un objet de la classe `string` et cet objet est initialisé à l'aide du constructeur à 1 argument de type `char*`.

```
ptr = new string("toto");      /* le constructeur à 1 argument de type char* de la classe
                                  string réalise  l'initialisation de l'objet */
```

10.2. Réalisation de la composition par gestion dynamique de mémoire (données en profondeur).

Nous avons vu dans la section 7 différentes réalisations de la composition. On peut également gérer un objet composant dans la zone de mémoire gérée dynamiquement (dans le tas). Dans ce cas, l'objet composant n'est pas réellement dans la classe mais est connu via un pointeur. Les données de l'objet composant n'étant plus dans l'objet composite, on dit que les données sont *en profondeur*, autrement dit obtenues indirectement par un pointeur. Si l'on reprend l'exemple de la classe `Point` qui est composée de 2 objets de type `Rationnel`, on peut donner une nouvelle réalisation de la composition basée sur la gestion dynamique d'objets (même si la représentation UML est la même)

```
-------------------------------------------------------------------------------
#ifndef _POINT__
#define _POINT__
#include "rationnel.h"

class Point
{
public:
      Point(const Rationnel & x, const Rationnel & y);
      Point(const Point & p);
      ~Point();     //destructeur
      Point & operator=(const Point & p);
      Rationnel GetX() const;
      Rationnel GetY() const;
private:
      Rationnel * _XY;      // pointeur sur le composant
};
#endif
-------------------------------------------------------------------------------
```



Remarque : pour la première fois depuis le début de ce manuel, le rôle du destructeur d'une classe va être illustré. Le destructeur est la méthode de la classe dont le nom est celui de la classe précédé d'un tilde (~), ci-dessous la méthode `~Point`. Cette méthode est appelée juste avant la disparition de l'objet.

```
-------------------------------------------------------------------------------
//Point.cpp
#include "Point.h"
#include <process.h>
#include <iostream>
using namespace std;

Point::Point(const Rationnel & x, const Rationnel & y){
      _XY=new Rationnel[2];
      if(_XY==NULL){
            cerr << "echec d'allocation de mémoire.";
            exit(2);
      }
      _XY[0]=x;
      _XY[1]=y;
}
// constructeur de copie
Point::Point(const Point &p){
      _XY=new Rationnel[2];
      if(_XY==NULL){
            cerr << "echec d'allocation.";
            exit(2);
      }
      _XY[0]=p._XY[0];
      _XY[1]=p._XY[1];
}
```

```
// destructeur : méthode appelée automatiquement juste avant la disparition de l'objet
Point::~Point()
{
	delete [] _XY;
}

Point & Point::operator=(const Point &p)
{
	_XY[0]=p._XY[0];
	_XY[1]=p._XY[1];
	return *this;
}

Rationnel Point::GetX() const {	return _XY[0]; }

Rationnel Point::GetY() const {	return _XY[1]; }
--------------------------------------------------------------------------------
```

Les données liées aux coordonnées ne sont pas directement dans l'objet `Point` mais dans une zone mémoire gérée dynamiquement par l'objet. Dans ce cas, le constructeur a pour rôle d'allouer de la mémoire et le *destructeur* (méthode `~Point`) celui de libérer la zone mémoire allouée par le constructeur.

Une telle structuration a des conséquences. Il faut écrire les constructeurs (qui allouent la mémoire), le destructeur (qui libère la mémoire) et l'opérateur d'affectation avec soin. Faute de quoi, des fuites mémoires peuvent avoir lieu.

10.3. Réalisation de conteneurs : tableaux, piles, files, listes ...

Les conteneurs sont des objets destinés à stocker/restituer des données. Les conteneurs se distinguent les uns des autres par la façon dont ils gèrent et accèdent aux données. Les conteneurs les plus utilisés sont :

- les tableaux : l'accès a une donnée est indicé et direct.
- les listes : l'accès à une donnée est séquentiel
- les piles/les files : on peut voir ces conteneurs comme des listes particulières

La taille des données d'un objet conteneur est susceptible d'évoluer durant la vie de l'objet. Un objet conteneur ne peut donc pas prévoir dès la construction quelle sera la quantité de mémoire qui lui sera nécessaire. Un conteneur va donc gérer dynamiquement la mémoire qui lui sera nécessaire. Les changements de taille des données conduisent à des allocations/désallocations de mémoire. Nous illustrons ceci par une formulation élémentaire d'une classe d'objets tableaux.

```
--------------------------------------------------------------------------------
#ifndef __CLS_TABLEAU__
#define __CLS_TABLEAU__

class Tableau
{
public:
	Tableau(unsigned taille=5);
	Tableau(const Tableau & tab);
	Tableau & operator=(const Tableau & tab);
	virtual ~Tableau();

	double & operator[](unsigned idx);
	double operator[](unsigned idx) const;

	void NouvelleTaille(unsigned taille);
	unsigned GetTaille() const;
```

```
        unsigned GetCapacite() const;
 private:
        double * _tab;
        unsigned _capacite;
        unsigned _taille;
 };
 #endif
 ---------------------------------------------------------------------------
```

La mémoire est gérée ici selon la technique utilisée dans la bibliothèque STL pour les `vectors`. Un objet alloue une zone mémoire d'une certaine capacité (attribut `_capacite`). Dans cette zone, il stocke les données utiles (qui occupent une certaine `_taille`). Par conséquent, `_taille` doit toujours être inférieur ou égal à `_capacite`. La différence `_capacite-_taille` constitue une réserve de mémoire utilisable lors des changements de taille ou des affectations entre objets tableaux.

```
 ---------------------------------------------------------------------------
 #include "Tableau.h"
 #include<process.h>
 #include<iostream>
 using namespace std;

 Tableau::Tableau(unsigned taille):_taille(taille),
                             _capacite(2*taille), _tab(new double[2*taille])
 {
if(_tab==NULL){
        cerr << "echec d'allocation";
        exit(2);
}
// les éléments du tableau sont initialisés à 0
for(unsigned i=0;i<_taille;i++) _tab[i]=0;
 }

 Tableau::~Tableau(){
        delete [] _tab;
 }

 // constructeur de copie
 Tableau::Tableau(const Tableau & t):_taille(t._taille),
                                     _capacite(t._capacite),_tab(new double[t._capacite])
 {
if(_tab==NULL){
        cerr << "echec d'allocation";
        exit(2);
}
for(unsigned i=0;i<_taille;i++) _tab[i]=t._tab[i];
 }

 double & Tableau::operator [](unsigned idx)
 {
if(idx>=GetTaille()){        // verification d'indice
        cerr << "indice incorrect!";
        exit(2);
}
return _tab[idx];
 }
```

```
 double Tableau::operator [](unsigned idx) const
 {
if(idx>=GetTaille())  {
       cerr << "indice incorrect!";
       exit(2);
}
return _tab[idx];
 }

 // opérateur d'affectation
 Tableau & Tableau::operator=(const Tableau & t)
 {
if(_capacite<t._taille) // réallocation si capacité insuffisante
{
       _capacite=2*t._taille;
       double * temp=new double[_capacite];
       if(_tab==NULL){
       cerr << "echec d'allocation";
       exit(2);
       }
       delete [] _tab;
       _tab=temp;
}
_taille=t._taille;
for(unsigned i=0;i<t._taille;i++) _tab[i]=t._tab[i];
return *this;
 }

 unsigned Tableau::GetTaille() const {
       return _taille;
 }

 unsigned Tableau::GetCapacite() const {
       return _capacite;
 }

 // méthode permettant le changement de taille
 void Tableau::NouvelleTaille(unsigned taille)
 {
if(taille<=GetTaille())      _taille=taille;
else
{
       if(taille>_capacite)
       {
              _capacite=2*taille;
              double * temp=new double[_capacite];
              if(_tab==NULL){
              cerr << "echec d'allocation";
              exit(2);
              }
              for(unsigned i=0;i<_taille;i++) temp[i]=_tab[i];
              delete [] _tab;
              _tab=temp;
       }

       for(unsigned i=_taille;i<taille;i++) _tab[i]=0;
       _taille=taille;
}
 }
 ---------------------------------------------------------------------------
```

Fonction retournant une référence

Il faut noter que l'opérateur d'indexation `[]` retourne une référence. Ainsi on peut modifier un élément du tableau via cet opérateur. Dans l'implémentation de l'opérateur simplifiée ici

```
 double & Tableau::operator [](unsigned idx){
return _tab[idx];
 }
```

il faut interpréter cela comme le fait que la fonction retourne une référence à `_tab[idx]`. Dans ce cas, la fonction ne retourne pas une copie de la case, mais la case elle-même.

Deux opérateurs d'indexation []

Il faut également remarquer que la classe dispose de deux opérateurs d'indexation. Un opérateur constant qui peut être appelé sur des objets constants (c'est-à-dire des tableaux constants). Celui-ci ne doit pas pouvoir modifier les éléments du tableau et renvoie donc le contenu de la case par valeur (retour classique). L'autre opérateur (non constant) quant à lui retourne une référence.

D'ailleurs, on ne peut appeler sur un objet constant que les méthodes constantes : ici il s'agit de `GetTaille()`, `GetCapacite()` et un opérateur `[]`. L'exemple ci-dessous montre comment l'on peut utiliser la classe `Tableau`.

```
 ------------------------------------------------------------------------------
 #include "Tableau.h"
 #include<iostream>
 using namespace std;
 void main()
 {
Tableau T1(4);
T1[0]=2;       // modifie la case d'indice 0
T1[2]=1.3;

cout << T1[0] << endl ;
cout << T1.GetTaille() << endl;

// ici T1 contient {2,0,1.3,0}, _taille==4 et _capacite==8

T1.NouvelleTaille(10);

// ici T1 contient {2,0,1.3,0,0,0,0,0,0,0}, _taille==10 et _capacite==20

const Tableau T2(T1); T2 est un tableau constant, fait par copie de T2

// ici T2 contient {2,0,1.3,0,0,0,0,0,0,0} et son état n'est pas censé évoluer
// on ne peut appeler sur T2 que des méthodes constantes

cout << T2[7] << endl ;
cout << T2.GetTaille() << endl;
 }
 ------------------------------------------------------------------------------
```

Cette classe ressemble, en beaucoup moins évoluée, à la classe `vector<double>` de la bibliothèque STL. Il reste néanmoins préférable d'utiliser la bibliothèque STL plutôt que de redéfinir des conteneurs. Cependant, écrire des conteneurs est un bon exercice pour assimiler les techniques d'écriture des classes. Il peut être intéressant de développer, à titre d'exercice, les conteneurs suivants (présents dans la bibliothèque STL): classe `Ensemble` (permettant de stocker des éléments sans doublon possible), classe `ListeChainee` (classe de listes chaînées), classe `Pile`(structure LIFO), classe `Chaine` (classe de chaînes de caractères).

11. Les fonctions et les classes paramétrées en type

On a vu que la bibliothèque STL fournit des classes de conteneurs paramétrées en type. Par exemple, on peut faire des tableaux ou des listes d'éléments de n'importe quel type. Nous allons préciser ici la syntaxe permettant de réaliser des classes ou des fonctions paramétrées en type.

11.1. Les fonctions paramétrées en type

Considérons l'exemple classique d'une fonction de permutation des valeurs de deux variables entières.

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

      void permute(int & a,int & b){ int c=a; a=b; b=c; }

void main()
{
      int x=2,y=3;
      permute(x,y);
      cout << x << y << endl;
}
-------------------------------------------------------------------------------
```

Nous voulons désormais, dans la même application, permuter deux valeurs de type `double`. Grâce à la surcharge de fonctions, on peut compléter le programme de la manière suivante

```
-------------------------------------------------------------------------------
      void permute(int & a,int & b){ int c=a; a=b; b=c;  }

      void permute(double & a, double & b) { double c=a; a=b; b=c; }

void main()
{
      int x=2,y=3;
      double u=1.3,v=1.7;
      permute(x,y);
      permute(u,v);
}
-------------------------------------------------------------------------------
```

La seule différence entre ces deux fonctions concerne les types utilisés. Si l'on peut paramétrer les types, on n'a qu'une seule définition à donner. C'est ce que permet la notation `template` (patron).

```
-------------------------------------------------------------------------------
      template<class T> void permute(T & a,T & b)
      {
            T c=a; a=b; b=c;
      }
void main()
{
      int x=2,y=3;
      double u=1.3,v=1.7;
      permute(x,y);  // T=int
      permute(u,v);  // T=double
}
-------------------------------------------------------------------------------
```

La notation **`template<class T>`** indique au compilateur que **`T`** est un paramètre de type (type primitif, structuré ou classe). La fonction permute peut alors être appelée avec deux arguments d'un type quelconque,

dans la mesure où les deux arguments sont du même type (ça n'aurait pas de sens de permuter un caractère avec un flottant).

Quand un paramètre de type est utilisé dans une fonction C++, on parle alors de *patron de fonctions*, ou de *fonction paramétrée en type* ou encore de *fonction générique*. Un patron de fonctions ne conduit pas directement à du code. Par exemple, s'il n'y avait pas d'appel de la fonction `permute()`, aucun code ne serait généré (contrairement à une fonction classique). Le code des fonctions créées à partir du patron n'est généré que s'il y a un appel de la fonction pour un type donné.

Dans l'exemple précédent, le compilateur crée deux fonctions à partir du patron. La première fonction est créée pour le paramètre de type `T=int` et la seconde pour le paramètre de type `T=double`. Le code des fonctions est généré selon le besoin. Dans un patron de fonctions, on peut aussi faire apparaître des arguments typés normalement. Ces paramètres sont parfois appelés *paramètres expression*. L'exemple suivant représente ce que pourrait être une fonction de tri de tableau paramétrée :

```
template <class T> void tri(T * tab,unsigned taille)
{
              // algorithme du tri d'un tableau de taille éléments de type T;
              // ... //
}
```

Un patron de fonctions définit une famille de fonctions ayant toutes le même algorithme mais pour des types différents. Il reste néanmoins possible de définir explicitement le code d'une fonction qui devrait normalement être prise en charge par le patron. Dans l'exemple suivant, nous réalisons un patron de fonctions calculant le minimum de deux variables de même type. La fonction `min()` obtenue à partir du patron a du sens pour tous les types élémentaires sauf pour le type `char*`. Nous avons donc défini une spécialisation pour le type `char *`. Cette spécialisation remplace le code que génèrerait le patron.

```
-------------------------------------------------------------------------------
#include<iostream>
#include<string.h>
using namespace std;

//patron de fonctions
template <class T> T min(T a,T b){ return a<b ? a : b; }

//spécialisation du patron pour le type char *
char * min(char * s1,char * s2)
{
      if(strcmp(s1,s2)<0)   return s1;
      else   return s2;
}

void main()
{
      int a=1,b=3;
      char chaine1[]="coucou",chaine2[]="salut";
      cout << min(a,b); //générée à partir du patron
      cout << min(chaine1,chaine2);  //spécialisation
}
-------------------------------------------------------------------------------
```

11.2. Classes paramétrées en type

On peut également introduire des paramètres de type dans la définition d'une classe. D'ailleurs, les conteneurs STL (voir section 6) sont des classes paramétrées en type. Pour présenter la syntaxe, le plus simple

est une fois encore de traiter un exemple. On va réécrire la classe `Point` de sorte de pouvoir instancier des points dont les coordonnées soient d'un type choisi par l'utilisateur.

On rappelle que dans l'utilisation d'une classe paramétrée, la valeur du paramètre de type est précisée entre `< T >`. Par exemple, pour la classe paramétrée `Point<T>` donnée ci-après, l'utilisation sera la suivante.

```
-------------------------------------------------------------------------------
void main()
{
	Point<int> p1(12,16);		// point à coordonées entières
	Point<double> p2(1.3,4.6);	// point à coordonnées rationnelles
}
-------------------------------------------------------------------------------
```

Définition de la classe paramétrée **`Point<T>`.**

```
-------------------------------------------------------------------------------
//fichier point.h
#ifndef __POINT_TEMPLATE__
#define __POINT_TEMPLATE__

template<class T>    // T est un paramètre de type
class Point
{
public:
	Point<T>(){ _XY[0]=0; _XY[1]=0;}

	Point<T>(const T & x,const T & y)
	{
		_XY[0]=x;
		_XY[1]=y;
	}

	Point<T>(const Point<T> & p)
	{
		_XY[0]=p._XY[0];
		_XY[1]=p._XY[1];
	}

	Point<T> & operator=(const Point<T> & p)
	{
		_XY[0]=p._XY[0];
		_XY[1]=p._XY[1];
		return *this;
	}

	void SetX(const T & x) { _XY[0]=x; }
	void SetY(const T & y){ _XY[1]=y; }

	const T GetX() const { return _XY[0]; }
	const T GetY() const { return _XY[1]; }

private:
	T _XY[2];	// tableau de 2 éléments de type T
};
#endif
-------------------------------------------------------------------------------
```

Remarque : pour les classes paramétrées en type, toute l'implémentation des méthodes doit être disponible quand on utilise la classe paramétrée. Aussi, il est habituel que dans ce cas, le corps des méthodes soit fourni

dans la définition de la classe. Il n'y a donc qu'un fichier header (fichier d'ntête d'extension .h) contenant toute l'implémentation des méthodes (comme ci-dessus). On doit aussi remarquer que la définition du paramètre de type n'est donnée qu'une seule fois avant le début de la classe.

12. Les flots entrée/sortie, les fichiers

Ce chapitre présente un aperçu très sommaire des flots et de l'utilisation des fichiers (vus comme des flots particuliers). On trouve une présentation détaillée de ces flots (modification du formatage, les manipulateurs de flots) dans différents livres sur le C++ (dont celui de C.Delannoy) ainsi que sur l'aide MSDN accessible sur http://msdn.microsoft.com/

12.1. Les déclarations d'énumérations au sein des classes

La déclaration d'énumérations s'effectue par la syntaxe suivante :

```
enum bool{false=0,true};
```

`bool` est un type, et `false/true` sont des valeurs pour le type `bool`. Les énumérations sont gérées comme des variables de type entier. En C++, on peut définir des énumérations au sein des classes. On définit ainsi une famille de constantes au sein d'une classe (ce qui est une alternative à la déclaration de constantes via la directive `#define`).

```
class Calendrier
{
public:
	enum jour{lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche};

	Calendrier();

	//...
};
```

Le type énumération, ainsi que ses valeurs possibles, appartiennent à la classe `Calendrier`. C'est donc l'opérateur de résolution de portée `::` qui permet de les référencer.

```
Calendrier::jour j=Calendrier::mardi;
```

Cette technique est très couramment utilisée, notamment dans la bibliothèque STL pour les modes d'ouverture des flots et la définition des formats.

12.2. La déclaration d'une classe au sein d'une classe

C'est la technique utilisée par les itérateurs dans la bibliothèque STL. La classe ci-dessous illustre cette technique qui peut se généraliser aux patrons de classes.

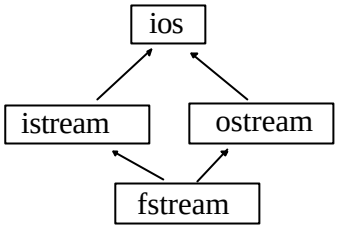
```
class vecteur{
public:
	class iterateur{
	public:
		iterateur();
		...
	};
	vecteur(int taille);
	iterateur begin() const;
	int & operator[](int idx);
};
```

Ici, la classe `iterateur` est définie dans la classe vecteur. L'utilisation est la suivante.

```
vecteur v1(3);
vecteur::iterateur i=v1.begin();
```

12.3. La hiérarchie des classes de flots

Une vue (très partielle) de la hiérarchie des classes de gestion des flots entrée/sortie est la suivante.



La classe `ios` est la classe de base pour tous les flots. La classe `istream` est destinée aux flots d'entrée et `ostream` aux flots de sortie. D'ailleurs, `cin` et `cout` sont des objets globaux instanciés à partir de ces classes. Enfin, la classe `fstream` permet la gestion des fichiers. On voit qu'elle dérive à la fois des deux classes `istream` et `ostream`.

Remarque : l'héritage multiple est possible en C++ mais n'est pas abordé dans ce cours.

Par conséquent, la gestion des fichiers via la classe `fstream` utilise des méthodes héritées des classes mères. C'est pourquoi on va survoler l'interface de ces classes.

`ios::open_mode` : énumération définie dans la classe `ios`. Associée en particulier aux modes d'ouverture des fichiers. Les valeurs possibles sont :

`ios::in` : ouverture en lecture

`ios::out` : ouverture en écriture

`ios::app` : ouverture en ajout (mode append)

`ios::binary` : ouverture en mode binaire. (Par défaut, on ouvre le fichier en mode texte.)

Ces modes peuvent être combinés par l'opérateur |, par exemple **`ios::out|ios::app`**.

Méthodes de la classe `ostream` :

`ostream::put(char)` : sortie non formatée d'un caractère

`ostream::write(char *,int n)` : sortie de n caractères

`ostream::operator<<( )` : insertion formatée dans le flot pour certains types de base (`char`, `int` ...)

```
#include<fstream>
#include<iostream>
using namespace std;
void main()
{
       cout.put('a');
       cout.put('\n');
       char str[]={'a','b','c',0,'f','g','h'};
       cout << str << endl;
       cout.write(str,7);
}
```

A titre d'exemple, le format de sortie peut être modifié par certaines méthodes de la classe ainsi que par des manipulateurs de flot (ci-dessous, `endl` est un manipulateur de flot)

```
void main()
{
	float f1=1.3698;
	cout << f1 << endl;   // sortie formatée -> 1.3698
	cout.precision(3);    // changement de la précision
	cout << f1 << endl;   // sortie formatée -> 1.37
}
```

Méthodes de la classe istream

`istream::get(char &)` : lecture d'un caractère

`istream::getline(char * pStr,int nbCarac,char delim='\n')` : lecture d'une suite d'au plus **`nbCarac`** caractères. La lecture s'arrête aussi si le caractère délimiteur (par défaut '\n') est rencontré.

`istream::read(char * pStr,int nbCarac)` : lecture de **`nbCarac`** caractères.

`istream::operator>>( )` : lecture formatée pour les types de base.

```
//... inclusion des fichiers ...
using namespace std;

void main()
{
	char tampon[30];
	string str;
	cin.getline(tampon,30,'\n'); //la lecture s'arrête sur retour chariot
	cin >> str;                  // la lecture s'arrête sur les espaces

	cout << tampon << endl;
	cout << str << endl;
}
```

12.4.Les fichiers

En C++, les fichiers peuvent être gérés par des flots de type `fstream` (file stream). La classe `fstream` hérite des méthodes des classes `istream` et `ostream`, donc de celles présentées précédemment. La classe `fstream` dispose des méthodes suivantes (ou héritées de `ios`, `istream`, `ostream`)

```
	fstream::fstream() : crée un objet non connecté à un fichier
	fstream::fstream(char * nomFichier, ios::open_mode ) : ouvre un fichier
	fstream::~fstream() : fermeture d'un fichier ouvert

	fstream::open(char *, ios::open_mode) : ouvre un fichier
	fstream::close() : ferme le fichier en cours
```

Quelques équivalences entre les modes d'ouverture des flots et les modes d'ouverture de fichiers par fopen() en C

ios::open_mode	mode pour la fonction fopen()
ios::out	w
ios::in	r
ios::out|ios::app	a
ios::in|ios::out	r+
ios::in|ios::binary	rb
ios::out|ios::binary	wb
ios::out|ios::app|ios::binary	ab

```
void main()
{
	char tampon[150];

	// création ou remplacement du fichier
	fstream f("toto.txt",ios::out);

	// écriture dans le fichier
	f<<"test \t" << 1 << "\t"<< 1.5 << endl;
	f.close();

	// réouverture du fichier en mode ajout
	f.open("toto.txt",ios::out|ios::app);
	f<<"ajout \t" << 2 << "\t"<< 2.5 << endl;   // écriture dans le fichier
	f.close();

	// ouverture en mode lecture
	f.open("toto.txt",ios::in);

	while(!f.eof())
	{
		f.getline(tampon,150);
		cout  << tampon << endl;
	}
}
```

12.5. L'accès direct dans les fichiers

La classe `fstream` hérite de méthodes de positionnement.

Pour la lecture (hérité de istream) :

```
fstream::seekg(déplacement, position) : positionne le pointeur
fstream::tellg() : retourne la position courante
```

Idem pour l'écriture (hérité de ostream) :

```
fstream::seekp(déplacement, position)
fstream::tellp()
```

La classe `ios` définit pour cela des valeurs `ios::beg` (début du flot), `ios::end` (fin du flot), `ios::cur` (position courante). Il est à noter que la lecture ou l'écriture font évoluer la position courante.

```
void main()
{
	char tampon[150];

	// ouverture en écriture
	fstream f("fichier.txt",ios::out);
	for(int i=0;i<3;i++)  f<<"ligne numero "<< i << "\n";
	// instant t1

	f.seekp(0,ios::beg);  //positionne en début
	f<<"premiere ligne \n";
	f.close();

	// instant t2
	// ouverture en lecture
	f.open("fichier.txt",ios::in);
```

```
buffer fichier.txt
(instant t1)
--ligne numero 0--
--ligne numero 1--
--ligne numero 2--
```

```
buffer fichier.txt
(instant t2)
premiere ligne
ro 0--
--ligne numero 1--
--ligne numero 2--
```

```
	//positionne au 3ième du début
	f.seekg(2*sizeof(char),ios::beg);
	f.getline(tampon,150);
	cout << tampon <<endl;

	// déplace de 3 caractères
	f.seekg(3*sizeof(char),ios::cur);
	f.getline(tampon,150);
	cout << tampon <<endl;
}
```

```
sortie
re ligne
0--
Press any key to continue
```

13. Les changements de type

13.1. Les constructeurs comme convertisseurs de type

Considérons la classe `Entier` suivante. Les objets de cette classe encapsulent un entier. Cette classe sert uniquement à mettre en évidence les conversions possibles grâce aux constructeurs.

```
-------------------------------------------------------------------
#ifndef __CLS_ENTIER__
#define __CLS_ENTIER__

class Entier
{
public:
	Entier();
	Entier(int val);
	Entier(float val);
	Entier & operator=(const Entier & e);
	int Get() const;
private:
	int _data;
};
	Entier operator+(const Entier & e1,const Entier & e2);
#endif
-------------------------------------------------------------------

-------------------------------------------------------------------
// Entier.cpp
#include "Entier.h"
#include<iostream>
using namespace std;

Entier::Entier()
{
	cout << "constructeur Entier::Entier()"<<endl;
	_data=0;
}
Entier::Entier(int val) {
	cout << "constructeur Entier::Entier(int)"<<endl;
	_data=val;
}
Entier::Entier(float val)
{
	cout << "constructeur Entier::Entier(float)"<<endl;
	_data=(int)val;
}
int Entier::Get() const
{
	return _data;
}
```

```
Entier operator+(const Entier & e1,const Entier & e2)
{
       cout << "operator+(Entier,Entier)"<<endl;
       return e1.Get()+e2.Get();
}

Entier & Entier::operator =(const Entier & e)
{
       cout << "Entier::operator=(Entier)"<<endl;
       _data=e._data;
       return *this;
}
-----------------------------------------------------------------
```

Le programme ci-dessous met en évidence le rôle des constructeurs à 1 argument pour la conversion de type.

```
-----------------------------------------------------------------
#include<iostream>
#include "entier.h"
using namespace std;

       void f(Entier e)
       {
              cout << "valeur de e = " << e.Get() << endl;
       }

void main(void)
{

       int varInt=4;
       float varFloat=2.5;

       f(varInt);     ;conversion int->Entier
       f(varFloat);   ;conversion float->Entier

       Entier e(3);

       e=e+varFloat; // conversion float->Entier

       e=1+e;  // conversion int->Entier

}
-----------------------------------------------------------------
```

Sortie

```
constructeur Entier::Entier(int)
valeur de e = 4
constructeur Entier::Entier(float)
valeur de e = 2
constructeur Entier::Entier(int)
constructeur Entier::Entier(float)
operator+(Entier,Entier)
constructeur Entier::Entier(int)
Entier::operator=(Entier)
constructeur Entier::Entier(int)
operator+(Entier,Entier)
constructeur Entier::Entier(int)
Entier::operator=(Entier)
```

La règle suivante s'applique en C++ : lorsqu'un objet de type classe `CLS` (ici `Entier`) est attendu, et qu'une variable `autre` de type `autreType` est fournie, le compilateur cherche si parmi les constructeurs de la classe `CLS` il en exite un qui accepte un seul argument de type `autreType`. Si un tel constructeur exite, un objet temporaire est construit avec ce constructeur et est utilisé à la place de la variable `autre`.

Autrement dit :

Les constructeurs à un argument définissent une règle de conversion.

Par exemple, le constructeur `Entier::Entier(float)` définit la conversion d'un type `float` vers le type `Entier`. Aussi, à chaque fois qu'un type `Entier` est attendu, et qu'un type `float` est fourni, le constructeur réalise un objet temporaire pour cette conversion. Ce rôle des constructeurs doit être bien compris car en pratique beaucoup de conversions de type sont réalisées *tacitement* sans que le programmeur en soit tenu informé.

On peut donner un autre exemple utilisant la classe `string` de la STL. Puisqu'il est possible de créer un objet `string` à partir d'une chaine de caractères du C, les lignes suivantes sont compilées sans aucun problème.

```
string s1("chaine");        // constructeur string::string(char *)
s1=s1+"ajout";              // conversion char * -> string avant appel de l'opérateur +
```

En effet, là où l'opérateur `+` attend un objet de type `string`, on lui passe une variable de type `char *`. Le compilateur cherche donc s'il est possible de convertir la chaîne `"ajout"` en un objet de type `string`. C'est possible car la classe `string` dispose d'un constructeur à un argument de type `char *`. La chaîne `"ajout"` est donc concaténée ici à la chaine `s1`. Grâce à celà, on peut utiliser des chaînes du C à la place des objets `string`.

Mot clé `explicit` : par défaut, les conversions possibles grâce aux constructeurs peuvent être réalisées implicitement (tacitement) par le compilateur (il n'y a pas de warning). Il peut donc y avoir des conversions sans que le programmeur en soit conscient. On peut néanmoins mentionner dans la classe que les constructeurs ne soient utilisés comme convertisseurs que de manière explicite (mot clé **`explicit`**).

```
class Entier
{
public:
      Entier();
      explicit Entier(int val);    // utiliable seulement pour conversions explicites
      explicit Entier(float val);
      ...
};
```

Dès lors, toutes les conversions `int->Entier` et `float->Entier` doivent être explicites.

```
#include<iostream>
#include "entier.h"
using namespace std;

      void f(Entier e){
            cout << "valeur de e = " << e.Get() << endl;
      }

void main(void){
            int varInt=4;
            float varFloat=2.5;
            f((Entier)varInt);           // conversion explicite
            f((Entier)varFloat);
            Entier e(3);
            e=e+(Entier)varFloat;
            e=(Entier)1+e;
}
```

13.2. Conversion d'un type classe vers un type primitif

On peut également définir des règles permettant de convertir un objet d'un type classe en un type de base. Il s'agit en fait de définir le traitement réalisé par un *opérateur de cast.* Dans la classe `complexe` suivante, l'opérateur `double` réalise la conversion `complexe->double`. Là où un objet `complexe` est passé alors qu'un `double` est attendu, cet opérateur est appelé pour réaliser la conversion de type.

```
-------------------------------------------------------------------------------
//complexe.h
#ifndef __COMPLEXE__
#define __COMPLEXE__
#include<iostream>
class complexe
{
private:
      double re;     // partie réelle
```

```
	double im;	// partie imaginaire
public:
	complexe(double r=0,double i=0){	// constructeur
		cout << "complexe(double=0,double=0) \t objet : "<<this << endl;
		re=r;
		im=i;
	}

	complexe operator+(const complexe & c) const
	{
		cout << "complexe::operator+() \t obj:"<<this<<" + obj:" <<&c<<endl;
		return complexe(re+c.re,im+c.im);
	}

	operator double(){	// opérateur de cast
		cout << "complexe::operator double() \t obj:"<<this<<endl;
		return re;	// la conversion complexe->double retourne la partie réelle
	}
	friend ostream & operator<<(std::ostream & flot,const complexe & c);
};
std::ostream & operator<<(std::ostream & flot, const complexe & c)
{
	flot << c.re;
	if(c.im){flot << "+"<<c.im<<"i";}
	return flot;
}
#endif
-------------------------------------------------------------------------------
```

Le programme d'utilisation suivant illustre les conversions possibles grâce à cette classe élémentaire. On y retrouve aussi les conversions réalisées par le constructeur de la classe `complexe`. Il faut noter qu'on peut utiliser le constructeur de la classe `complexe` avec un seul argument de type `double`, l'autre ayant la valeur 0 par défaut. Ce constructeur peut donc être utilisé pour réaliser les conversions `double->complexe`.

```
-------------------------------------------------------------------------------
#include "complexe.h"

void main()
{
complexe c1,c2(1,3); //1)-2)

c1=2.4;              //3)
cout << c1 << endl;  //4)

c2=c1+c2;            //5)-6)
cout << c2 << endl;  //7)

c1=c1+(complexe)3.5; //8)-9)-10)
cout << c1 << endl;  //11)

c1=(double)c1+3.5;   //12)-13)
cout << c1 << endl;  //14)

double d;
d=c1;                //15)
cout << d << endl;   //16)

// c1=c1+3.5; // << error c2666 : 2 overloads have similar conversions
}
-------------------------------------------------------------------------------
```

```
Résultats
complexe(double=0,double=0)      objet : 0x0066FDDC       1)
complexe(double=0,double=0)      objet : 0x0066FDCC       2)
complexe(double=0,double=0)      objet : 0x0066FDB4       3)
2.4                                                       4)
complexe::operator+()    obj :0x0066FDDC + obj :0x0066FDCC 5)
complexe(double=0,double=0)      objet : 0x0066FDA4       6)
3.4+3i                                                    7)
complexe(double=0,double=0)      objet : 0x0066FD94       8)
complexe::operator+()    obj :0x0066FDDC + obj :0x0066FD94 9)
complexe(double=0,double=0)      objet : 0x0066FD84       10)
5.9                                                       11)
complexe::operator double()      obj :0x0066FDDC          12)
complexe(double=0,double=0)      objet : 0x0066FD74       13)
9.4                                                       14)
complexe::operator double()      obj :0x0066FDDC          15)
9.4                                                       16)
```

Les lignes de la sortie écran ont été numérotées : il y a 16 sorties écran.

1) construction de `c1` (objet d'adresse 0x0066FDDC)
2) construction de `c2` (objet d'adresse 0x0066FDCC)
3) construction d'un objet temporaire, à partir de l'argument 2.4, qui est passé à l'opérateur = . L'adresse de cet objet temporaire est 0x0066FDB4.

Note : puisque l'opérateur = n'a pas été explicitement défini, c'est l'opérateur = par défaut qui est utilisé. Ce dernier fait une copie membre-à-membre qui convient parfaitement puisque les objets de la classe `complexe` n'ont pas de données en profondeur.

4) Affichage du contenu de `c1` grâce à l'opérateur <<
5) L'opérateur + de la classe `complexe` est appelé.
6) Un objet temporaire d'adresse 0x0066FDA4 est construit pour retourner le résultat de l'opérateur car c'est un retour par valeur. C'est d'ailleurs cet objet temporaire qui est utilisé comme argument de l'opérateur =.
7) Affichage de `c2`
8) Lié à la conversion explicite. La valeur 3.5 doit être convertie en un objet `complexe`. Il y a donc construction d'un objet de la classe `complexe` d'adresse 0x0066FD94.
9) Appel de l'opérateur + de la classe `complexe`. On remarque que l'argument de l'opérateur est bien l'objet d'adresse 0x0066FD94
10) L'opérateur + retourne un objet par valeur. Il y a construction d'un objet d'adresse 0x0066FD84.
11) Affichage de `c1`
12) Là encore, il ya une conversion explicite. On souhaite convertir `c1` en type `double`, ce qui fait appel à l'opérateur `double` de la classe `complexe` sur l'objet `c1` (dont l'adresse est 0x0066FDDC)

Note : par conséquent, c'est l'opérateur + correspondant au type `double` qui est utilisé ici, et non celui du type `complexe`.

13) Puisque `(double)c1+3.5` est de type `double`, il doit y a voir une conversion implicite `double-> complexe` avant l'affectation. Il y a donc construction d'un objet temporaire d'adresse 0x0066FD74.
14) Affichage de `c1`
15) Il y a conversion implicite du type `complexe` vers le type `double`. Cette conversion implicite utilise l'opérateur `double` de la classe `complexe` sur l'objet `c1`.
16) Affichage de `c1`

Conclusion

Ces conversions de type constituent un sujet sensible. L'utilisation de ces moyens de conversions peut conduire à des situations où le compilateur ne sait pas quelle conversion mettre en œuvre. C'est le cas pour la ligne suivante :

```
c1=c1+3.5;    // c1=c1+(complexe)3.5 ?  ou c1=(complexe)((double)c1+3.5) ?
```

Le compilateur ne sait pas si l'on souhaite que 3.5 soit converti en `complexe` ou bien que `c1` soit converti en `double`. Le compilateur indique alors le message d'erreur : *error c2666 : 2 overloads have similar conversions.* Notons que cela ne conduit d'ailleurs pas au même résultat !

14. Conclusion et fiches synthétiques

Certains aspects du C++ ont été présentés dans ce document, mais la présentation faite ici est loin d'être exhaustive. Les notions suivantes ont été laissées de côté pour cette première approche du C++ : l'héritage multiple, les nouveaux opérateurs de cast (notamment dynamic_cast), le RTTI (Run Time Type Information), l'utilisation détaillée de la bibliothèque STL.

Pour finir, les fiches suivantes synthétisent certaines parties de ce cours.

Le C++ en dehors des classes

Les références En C++, les références sont des alias : plusieurs noms pour une même variable, un même objet.

Déclaration d'une référence

```
int & refI=varI;      //   refI est un alias de la variable varI
//  refI et varI sont deux identificateurs pour la même variable de type int
```

Deux alias fournissent la même adresse

```
int * p1=&refI;       // référence à varI
int * p2=&varI;
cout << p1 << p2 << endl;     // les deux pointeurs contiennent la même adresse
```

Ne pas confondre les deux sens de l'opérateur & : adresse de / déclaration d'une référence

```
int & refI=varI;      // déclaration d'une référence
int * p=&varI; // opérateur qui fournit l'adresse d'une variable
```

Paramètre "passé à une fonction par référence"

```
// utilisation de la fonction
f(pAppel);    //à la sortie de la fonction, le paramètre d'appel a pu être modifié

// définition de la fonction
void f(int & pFormel)// pFormel est ici un paramètre de sortie
{
      // le paramètre formel étant un alias du paramètre d'appel ...
      pFormel++;
      // equivaut à pAppel++
}
```

Paramètre "passé à une fonction par référence constante"

```
void f(const GrosType & pEntree) // pEntree est ici un paramètre d'entrée
{
      // si le paramètre est un gros (en nombre d'octets) paramètre d'entrée,
// le passage par référence est plus rapide, mais il faut alors ajouter le mot clé const
// pour prévenir toute modification fortuite.
}
```

Les paramètres en C++ (synthèse)

Un *petit paramètre* (char, int, double, etc.) d'entrée ou d'entrée/sortie peut être *passé par valeur* (ou *par valeur constante*.)

Un *gros paramètre* (objet ou structure avec beaucoup de données) d'entrée ou d'entrée/sortie doit être passé *par référence* constante pour des raisons de performance.

Un paramètre de *sortie* doit être passé *par référence* (remplace le passage par adresse du langage C)

En C++, les tableaux sont passés aux fonctions comme en langage C, c'est-à-dire en fournissant l'adresse de leur première case et leur taille.

```
void f(int * tab, unsigned taille){ }
```

Retour d'une fonction par référence

```
int & f(int pF){ ...}
f(3)=2;// f(3) est une référence à une "certaine" variable, et est donc modifiable.
```

Surcharge de fonctions Plusieurs fonctions peuvent avoir le même nom mais pas la même signature

Paramètres avec valeurs par défaut Valeur attribuée au paramètre quand celui-ci fait défaut

```
void f(int pI, double d=2.1);
void f(char *, int i=4);
```

Les classes en C++

Déclaration d'une classe (dans un fichier header d'extension .h)

```
class MaClasse
{
	public:
		// méthodes à accès public
		MaClasse();			// constructeur sans argument
		MaClasse(int var);		// constructeur à un argument de type int
		MaClasse(const MaClasse & obj);	// constructeur de copie
		~MaClasse();			// destructeur

		MaClasse & operator=(const MaClasse & obj); // opérateur d'affectation

		void SetVal(int val);  // Méthode modificateur d'état, donc non constante

		int GetVal() const;   // Méthode pour obtenir tout ou partie de l'état
				      // donc méthode constante
private:
		// données / méthodes à accès privé
		int _val;		// données membres encapsulées (à accès privé)
		double _autre;
};
```

Définition des méthodes dans un fichier séparé (fichier d'extension .cpp) [source partielle]

```
MaClasse::MaClasse()
{
	_val=0;		// la donnée membre _val est initialisée à 0;
	_autre = 2.1;
}

MaClasse::MaClasse(const MaClasse & obj)
{
	// implementation non fournie
	...
}

int MaClasse::GetVal() const
{
	return _val;  // retourne la valeur de la donnée membre _val(l'état de l'objet)
}

void MaClasse::SetVal(int val) const
{
	_val=val;	// affecte une nouvelle valeur à l'état
}
```

A noter

- un constructeur/un destructeur ne retourne rien (ne pas mettre void non plus)
- le destructeur n'a pas d'argument
- un constructeur sert à donner un état cohérent à l'objet (penser à initialiser toutes les données membres)
- un accesseur est une méthode qui retourne ou donne accès à l'état de l'objet. C'est une méthode constante.
- inversement, une méthode qui permet la modification de l'état d'un objet ne doit pas être constante.

Utilisation de la classe MaClasse

```
void main()
{
	MaClasse objet1;	// construction d'un objet à l'aide du
	// constructeur sans argument
	// etat = (objet1._val==0, objet1._autre==2.1)

	MaClasse objet2(15);  // utilise le constructeur avec 1 argument int pour objet2

	MaClasse objet3(objet1); // utilise le constructeur par copie pour objet3

	cout << objet1.GetVal() << endl;

	objet1.SetVal(12);    // nouvel état : (objet1._val==12,objet1._autre==2.1)
}
```

Gestion des projets avec des classes en C++

Pour chaque classe, on place dans un fichier header (extension .h) la définition de la classe, c'est-à-dire son nom, les noms et types des arguments et les prototypes des méthodes.

On place dans un fichier source (extension cpp) la définition des méthodes.

Enfin, pour utiliser la classe, il faut créer un projet incluant le fichier source d'utilisation de la classe (fichier cpp) et le (ou les) fichier(s) de définition des méthodes. Le fichier header doit être visible par tous les fichiers qui l'utilisent

Fichier header (fichier d'entête)

```
-------------------------------------------------------------------------
// fichier d'entête maclasse.h
#ifndef __MA_CLASSE__
#define __MA_CLASSE__        // directives pour compilation conditionnelle

#include <iostream>  // inclusion des fichiers nécessaires ici

class MaClasse
{
       public:
              MaClasse();

              void Affiche(std::ostream & sortie) const;
};
#endif
-------------------------------------------------------------------------
```

Fichier de définition des méthodes (implémentation de la classe)

```
-----------------------------------------------------------------------------------------------
// fichier maclasse.cpp
#include "maclasse.h"  // le fichier header doit être dans le même répertoire que ce source

MaClasse::MaClasse()
{
      //...
}

void MaClasse::Affiche(std::ostream & sortie) const
{
      sortie << "Affichage sur sortie standard";
}
-----------------------------------------------------------------------------------------------
```

Fichier d'utilisation de la classe

```
-----------------------------------------------------------------------------------------------
// utilisation.cpp
#include "maclasse.h"       // le fichier header doit être dans le même répertoire que ce
source

void main()
{
      MaClasse m;
      m.Affiche(std::cout);        //affiche sur flot cout
}
-----------------------------------------------------------------------------------------------
```

Projet C++ : pour pouvoir compiler et éditer les liens, les deux fichiers `maclasse.cpp` et `utilisation.cpp` doivent être dans un même projet.

Note : on peut également faire un fichier librairie à partir d'un ou de plusieurs fichiers de sources de définition des méthodes d'une classe. Pour utiliser une classe, l'utilisateur n'a plus ensuite qu'à créer un projet incluant la librairie (à la place des fichiers sources de définition des méthodes).

Introduction aux classes standard et aux classes paramétrées en type

Les classes standards telles que `istream`, `ostream` (pour les objets `cin` et `cout`), `string` utilisent un espace de nom particulier :

```
#include <iostream>          // déclaration des classes de flots
#include <string>            // déclaration de la classe de chaînes de caractères

using namespace std;         // utiliser l'espace de nom std

void main()
{
      cout << "Hello world !" << endl;

      string str1("abc");   // construction d'un objet string
      string str2("def");
      string str3(str1);    // str3 est construit par copie de str1

      str1=str1+str2;       // l'opérateur + réalise une concaténation

      cout << str1 << endl; // un objet string s'affiche sur le flot cout

      cout << "3ième caractère de str1 :"
cout << str1[2] << endl; // un objet string s'utilise comme un tableau
}
```

Utilisation d'une classe paramétrée en type.

La bibliothèque STL fournit des classes paramétrées en type pour stocker des données

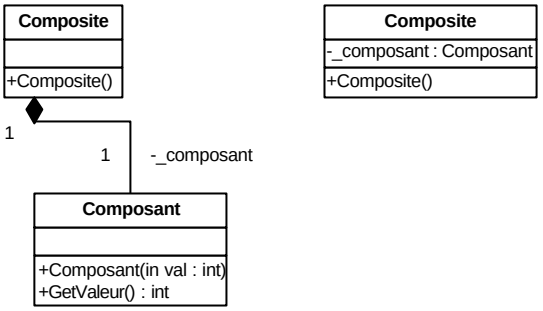
```
      vector<Ty>     = classe de tableaux stockant des données de type Ty
      stack<Ty>      = classe de piles de données de type Ty
```

```
-------------------------------------------------------------------------------------------
#include <iostream>
#include <vector>            // déclaration de la classe paramétrée vector<Ty>

using namespace std;         // utiliser l'espace de nom std aussi pour ces classes

void main()
{
      vector<int> v1(5,2);  // v1 est un vecteur de 5 int initialisés à 2.
      vector<int> v2(8);    // v2 est de taille 8, à valeurs initiales nulles

      v1[2]=3;       // v1 s'utilise comme un tableau
      v1[4]=2;

      // ici, éviter v1[5]=7 car les indices valides vont de 0 à 4 (car la taille est 5)

      v1.resize(12);        // méthode qui modifie la taille du vecteur

      cout << v1.size() << endl;   // vector<Ty>::size() est l'accesseur de taille

      v1[11]=4;      // ici c'est bon, car le vecteur vient d'être redimensionné

      v2=v1;         // l'affectation entre deux objets vector<int> est possible

      vector<char> v3(7);   // v2 est un vecteur de caractères de taille 7

      v2[0]='a';     // v3 stocke donc des caractères

      stack<int> pile;      // objet "pile" pour gérer des int

      pile.push(12);        // place la valeur 12 sur la pile

      stack<float> pileF;   // objet "pile" pour gérer des float

      pileF.push(3.25);     // place la valeur float 3.25 sur la pile pileF

      /* ce qui ne fonctionne pas (voir chapitre dédié aux classes paramétrées)

      v3=v1;         // les classes vector<int> et vector<char> sont différentes

      */
}
-------------------------------------------------------------------------------------------
```

Réalisation de la relation de composition en C++

Certaines classes ont comme attributs des objets d'une autre classe. On représente cela en UML soit par la relation de composition ou bien comme le fait que certains attributs sont d'un type classe donné.

Ci- contre, deux représentations de la même classe `Composite` sont données. La représentation de gauche indique que la classe `Composite` est composée d'un objet `_composant`. La représentation de droite indique qu'un attribut (`_composant`) de la classe `Composite` est de type `Composant`.



Dans les deux cas, une réalisation possible en C++ de la classe `Composite` est la suivante

```
-------------------------------------------------------------------------
//fichier composite.h
#ifndef __CLS_COMPOSITE__
#define __CLS_COMPOSITE__

#include "composant.h"

class Composite
{
public:
	Composite();
	Composant GetComposant() const;  // retourne une copie du composant
private:
	Composant _composant;		// objet membre
};
#endif
-------------------------------------------------------------------------
```

Définition des méthodes dans un fichier séparé (fichier d'extension .cpp)

```
-------------------------------------------------------------------------
#include "composite.h"
Composite::Composite():_composant(12)
{
}

Composant Composite::GetComposant() const
{
	return _composant;
}
-------------------------------------------------------------------------
```

Il faut souligner le rôle de la *liste d'initialisation* pour les classes composites. Ci-dessus, l'objet composant est initialisé par le constructeur à 1 argument de type int (seul constructeur disponible pour cette classe).

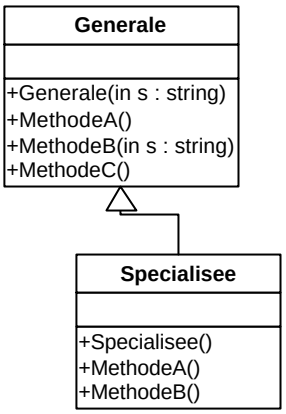
```
-------------------------------------------------------------------------
#include "Composite.h"

void main()
{
	Composite c1;

	Composant c2(c1.GetComposant());

}
-------------------------------------------------------------------------
```

Remarque : pour l'exemple ci-dessus, la classe Composant doit être dotée d'un constructeur de copie valide : d'une part pour que l'accesseur Composite::GetComposant() retourne une copie du composant et que la construction de c2 soit possible dans le programme d'utilisation.

Réalisation de la relation de spécialisation en C++

Une classe décrit ce que les objets d'un ensemble ont en commun en termes de méthodes et d'attributs. D'un point de vue ensembliste, une sous-classe caractérise donc un sous-ensemble d'objets. Soit cette sous-classe a des attributs supplémentaires permettant d'apporter une représentation plus précise des objets, soit elle se distingue par le fait que les attributs ne peuvent prendre que certaines valeurs. La relation sous-classe/super-classe ou relation de spécialisation/ généralisation se représente en UML par une flèche. La flèche part de la sous-classe (ou classe spécialisée) et désigne la super-classe (la classe la plus générale).

```
-------------------------------------------
// fichier header de la classe Specialisee
#ifndef __CLS_SPECIALISEE__
#define __CLS_SPECIALISEE__

#include "generale.h"

class Specialisee : public Generale
{
public:
       Specialisee();
       void MethodeA();
       void MethodeB();
};

#endif
-------------------------------------------
```



Définition des méthodes dans un fichier séparé (fichier d'extension .cpp)

```
-----------------------------------------------------------
// définition des méthodes de la classe Specialisee
#include "specialisee.h"
Specialisee::Specialisee():Generale(const std::string & s)
{
}

void Specialisee::MethodeA()
{
       // masque la méthode de la super-classe
       // ...
}
-----------------------------------------------------------
```

Là encore, il faut souligner le rôle de la liste d'initialisation. Ci-dessus, l'objet spécialisé initialise la partie héritée par le constructeur à 1 argument de type string. On peut supposer qu'ici la classe générale contient un objet membre de type string, mais ça importe peu. Ce qui importe c'est de regarder l'interface de la super-classe pour savoir quels sont les constructeurs disponibles.

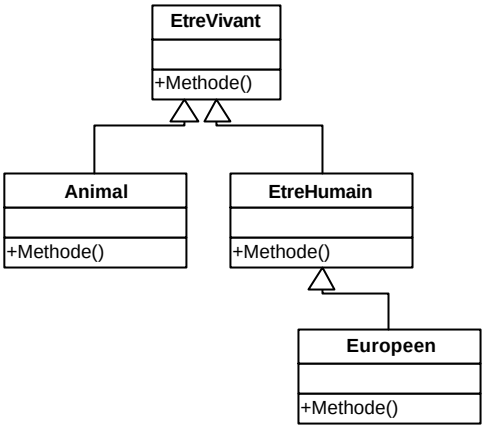
Toutes les méthode publiques de la super-classe sont utilisables sur la sous-classe. Eventuellement, certains phénomènes de masquage nécessitent l'utilisation du nom de la classe et de l'opérateur de portée ::

```
-----------------------------------------------------------
#include "specialisee.h"

void main()
{
       Specialisee s1;

       s1.MethodeC();  // la méthode C est accessible pour la sous-classe

       s1.MethodeB();        // méthode B de la super-classe
       s1.MethodeB(std::string("toto"));    // méthode B de la sous-classe

       s1.MethodeA();        // la méthode de la classe spécialisée
                             // masque celle de la super-classe

       s1.Generale::MethodeA();      // en raison du masquage, il faut préciser
                                     // si l'on souhaite
                                     // utiliser la méthode de la super-classe.
}
-----------------------------------------------------------
```

Le polymorphisme en C++

Lorsqu'une hiérarchie de classes existe, le C++ autorise certaines conversions de type. D'un point de vue ensembliste, il est normal que les objets d'une sous-classe puissent être aussi considérés comme des objets de la super-classe. Par exemple, la classe des êtres humains étant un sous-ensemble de celle des êtres vivants, un être humain est un être vivant.

```
class EtreVivant
{
public:
	void Methode();
};

class Animal: public EtreVivant
{
public:
	void Methode();
};

class Europeen: public EtreHumain
{
public:
	void Methode();
};
```



Pour la hiérarchie décrite ci-dessus, certaines conversions de types sont possibles. Par ailleurs, faute de précision, la ligature des méthodes est *statique*. Cela signifie que via un pointeur de type `EtreHumain *`, on n'accède qu'au méthodes de la classe `EtreHumain` ou des super-classess (par exemple `EtreVivant`).

```
void main()// conversion sous-classe * -> classe *
{
Europeen E;
Animal A;
EtreHumain * ptrEH=&E;	// Europeen * -> EtreHumain *

ptrEH->Methode();	// appel de la méthode de la classe EtreHumain même si l'objet
			// pointé est de type Europeen

ptrEH->EtreVivant::Methode();	//méthode également accessible

EtreVivant * ptrEV = ptrEH;  // conversion EtreHumain * -> EtreVivant *

ptrEV->Methode();	// appel de la méthode de la classe EtreVivant

EtrVivant * ptrG2 = &A;	// Animal * -> EtreVivant *
}
```

Ligature dynamique : on peut préciser dans la classe de base si une recherche dynamique de méthode doit être effectuée (mot clé virtual).

```
class EtreVivant
{
public:
virtual void Methode(); // demande au compilateur de mettre en place un lien dynamique
virtual ~EtreVivant();	// le destructeur doit être virtuel
};

void main()
{
	EtreVivant * tableau[4];
	tableau[0]=new EtreVivant;
	tableau[1]=new Animal;
	tableau[2]=new Europeen;
	tableau[3]=new EtreHumain;

tableau[0]->Methode();	// méthode de la classe EtreVivant
tableau[1]->Methode();	// méthode de la classe Animal
tableau[2]->Methode();	// méthode de la classe Europeen
tableau[3]->Methode();	// méthode de la classe EtreHumain

for(unsigned i=0;i<4;i++) delete tableau[i];
}
```

Les patrons de classes en C++

En C++, on peut introduire des paramètres de type. Pour les classes paramétrées, il est préférable de placer la définition des méthodes (elles-mêmes paramétrées) dans la définition de la classe. Le fichier header (.h) contient alors toute l'implémentation. Il n'y a pas de fichier de définition cpp pour les patrons de classe.

Exemple d'un patron de conteneurs de type tableau (accès direct).

```
-------------------------------------------------------------------------------
#ifndef __CLSPARAM__
#define __CLSPARAM__

template<class T>
class Tableau
{
public:
	Tableau<T>(unsigned size=1)
	{
		_tab=new T[size];
		_size=size;
	}

	Tableau<T>(const Tableau<T> & t)
	{
		_tab=new T[t._size];
		_size=t._size;
		for(unsigned i=0;i<_size;i++) _tab[i]=t._tab[i];
}

Tableau<T>& operator=(const Tableau<T> & t)
	{
		T* ptr=_tab;
		_tab=new T[t._size];
		_size=t._size;
		for(unsigned i=0;i<_size;i++) _tab[i]=t._tab[i];
		delete [] ptr;
		return *this;
}

~Tableau<T>(){ delete [] _tab;}

void GetElement(unsigned i) const { return _tab[i];}
void SetElement(unsigned i, const T & valeur) { _tab[i]=valeur; }

unsigned GetSize() const { return _size;}

T& operator[](unsigned i) {  return _tab[i]; }
const T &  operator[](unsigned i) const { return _tab[i]; }

private:
	T * _tab;		// T est le paramètre de type
	unsigned _size;
};
#endif
-------------------------------------------------------------------------------
```

Utilisation du patron :

```
-------------------------------------------------------------------------------
#include "Tableau.h"
#include "Rationnel.h"
#include "Point.h"


void main()
{
	Tableau<int> tab1(3);
	Tableau<Rationnel> tab2(4);
	Tableau<Point> tab3(10);

	tab1.SetElement(2,3);
	tab2.SetElement(0,Rationnel(1,2));
	tab1[0]=4;
}
-------------------------------------------------------------------------------
```