1. Rappels sur la POO en C++

1.1 Un peu d'histoire du C++

— Langage développé par Bjarne Stroustrup au cour des années 80 comme extension du langage C (C++ = C + extension non OO + extension OO).
— Première version en 1983 : fonctions virtuelles, surcharge des opérateurs et fonctions, contrôle de type amélioré, ...
— Version 2.0 en 1989 :héritage multiple, classes abstraites, membres protégés ... etc
— Standardisation ISO : 1998.
— Standard corrigé : 1993.
— Évolution de la Bibliothèque standard avec le langage (flux, STL ... etc).
— Langage libre, norme propriétaire

1.2 Extensions non OO dans C++

1.2.1 Les espaces de noms

C++ introduit les espaces de noms qui permettent de limiter la porté des identificateurs afin d'éviter les collisions de noms. Les bibliothèques standards du C++ sont organisées dans des espaces de noms disjoints (exemple l'espace std). L'utilisateur peut créer son propre espace de noms par :

```
1      namespace <nomespace>
2      {
3      Variables, fonctions, classes etc
4      } // ici se termine l'espace de noms.
```

Pour utiliser les éléments de l'espace précédent dans un programme C++ on utilise l'instruction : using namespace <nomespace>; supposons qu'une classe Point est défini dans le namespace A, l'instruction :

```
1      using A::Point
```

signifie que Point dans ce qui suit référence la classe Point définit dans l'espace de nom A.

R dans le cas de plusieurs instructions (**using namespace**), il faut faire attention aux ambigüités (collision de plusieurs éléments avec le même nom appartenant à des namespace différents) .

1.2.2 Nouvelles Entrées/Sorties

C++ offre le mécanisme de flux qui repose sur la surcharge des opérateurs « et » pour effectuer des E/S, Deux flux standards sont définis :

1. **cin :** entrée standard rattachée au clavier par défaut,
2. **cout :** sortie standard rattachée à l'écran par défaut.

```
1  #include <iostream>
2  using namespace std; // std contient les deux flots cin et cout
3  int main(){
```

```
4  int a;
5  cout <<"Donnez un nombre:" ;
6  cin >> a;
7  cout <<" Vous avez entrée le nombre:" << a<<" ,Merci";
8  }
```

1.2.3 Quelques autres extensions

1. Commentaires en ligne // ceci est un commentaire
2. Surcharge des fonctions (même nom, différents paramètres).
3. Passage des paramètres par référence : en C, le passage par adresse est simulé par l'utilisation des pointeurs, C++ dispose de la notion de référence réelle :

```
1  // en C :
2  void echange (int *a,int *b){
3       int c=*a;
4       *a=*b;
5       *b=c;
6  }
7  // en C++ :
8  void echange (int &a,int &b){
9       int c=a;
10      a=b;
11      b=c;
12 }
```

1.3 Concepts de l'AOO en C++

1.3.1 Éléments de base de la POO

— En programmation fonctionnelle : Une application est un ensemble de fonctions et de modules
— En POO : Une application est un ensemble d'objets en interaction par envoi de messages

Definition 1.3.1 Objet : c'est une représentation d'une entité du monde réel, ou du monde virtuel (dans le cas d'objets immatériels), qui se caractérisent par un **identificateur**, des **états** significatifs et par un **comportement**.

Definition 1.3.2 Identificateur d'un objet : Un descripteur qui permet de distinguer les objets entre-eux.

Definition 1.3.3 Attribut : (appelé aussi membre donnée en C++) Caractéristique d'un objet ex : taille d'un homme, surface d'une forme, etc.

Definition 1.3.4 L'état d'un objet : Correspond aux valeurs de tous ses attributs à un instant donné.

Definition 1.3.5 Le comportement d'un objet réside dans l'ensemble des opérations qu'il peut exécuter en réaction aux messages envoyés par les autres objets (un message = demande d'exécution d'une opération).

Definition 1.3.6 Cycle de vie d'un objet : Un objet est crée (statiquement / dynamiquement), utilisé puis détruit

Un objet peut être définit par $Objet =< Identificateur, Etat, Comportement >$

Definition 1.3.7 Une Classe d'objets est l'abstraction d'un ensemble d'objets qui possèdent une structure (liste des attributs) et un comportement (liste des opérations appelées méthodes) identiques

R Dans les codes illustratifs utilisés ultérieurement et pour une meilleure structuration, on sépare la déclaration des classes de leur implémentation, donc on écrit toujours 3 fichiers distincts :

- Un fichier header (.h) : contenant la déclaration des classes.
- Un 1er fichier .cpp : contenant l'implémentation de la classe.
- Un 2ème fichier .cpp contenant le code de test de la classe.

Exemple

on veut modéliser en C++ les nombres complexes comme des objets, on écrit la classe Complex :

— Fichier Complex.h :

```
class Complex {
    public:
        float rel;    //partie réelle du complexe
        float img;  // partie imaginaire
    public:
        void print();    // méthode d'affichage
        Complex(float,float);   // Constructeur paramétré
};
```

— Fichier Complex.cpp :

```
#include <iostream>
#include "Complex.h"
using namespace std;
void Complex::print(){
    cout<<rel<<" + "<<img<<"i\n";
}
Complex::Complex(float a,float b){
    rel=a;
    img=b;
}
```

— Fichier main.cpp :

```
#include <iostream>
```

```
2 #include "Complex.h"
3 using namespace std;
4 void Complex::print(){
5     cout<<rel<<" + "<<img<<"i\n";
6 }
7 Complex::Complex(float a,float b){
8     rel=a;img=b;
9 }
```

Les constructeurs en C++

Definition 1.3.8 Les Constructeurs sont des méthodes spécifiques dans une classe, appelés automatiquement lorsqu'on crée des objets. Ils portent le même nom que leur classe. Leur rôle est généralement l'initialisation des attributs de l'objet.

Definition 1.3.9 Un constructeur par défaut est un constructeur sans paramètres, si la classe ne comporte aucun constructeur, un par défaut est ajouté à cette classe avec une liste d'instructions vide.

Definition 1.3.10 Un constructeur par copie est un constructeur qui est invoqué lorsqu'on affecte un objet à un autre, il sert à créer un objet à partir d'un autre (clonage), l'objet à recopier doit être passer par référence à ce constructeur.

Dans l'exemple précédent, pour ajouter un constructeur par copie :

— Dans Complex.h on ajoute la déclaration du nouveau constructeur dans la classe :

```
1 void print();    // méthode d'affichage
2 Complex(float,float);  // Constructeur paramétré
3 Complex(Complex &); // Constructeur par copie
```

— Dans Complex.cpp on ajoute l'implémentation du nouveau constructeur

```
1 Complex::Complex(Complex& clone){
2     cout<<"Clonage ....\n"; rel=clone.rel; img=clone.img;
3 }
```

— Pour tester le constructeur par copie, dans le fichier main.cpp on ajoute :

```
1 Complex c2=c1;
2 c2.print();
```

R La copie est superficielle, si l'objet comporte un attribut alloué dynamiquement, le clone partagera cet attribut avec l'objet copié. Il faut s'assurer que les copies soient complètement disjointes dans le constructeur par copie.

Definition 1.3.11 Le Destructeur est une méthode particulière invoquée lorsque on détruit les objets (fin de portée pour les objets statiques).

Si on veut insérer un destructeur dans notre classe : Dans Complex.h :

```
~Complex();
```

Dans Complex.cpp :

```
Complex::~Complex() {
    cout<<" Au revoir ....\n";
}
```

Les membres de classes(statiques)

certains attributs et méthodes ont une signification pour la classe et non pour chaque objet(instance) à part, par exemple le nombre d'instances d'une classe est un attribut statique. Les attributs/méthodes statiques sont déclarés par le mot clé static et doivent être initialisés en dehors de la déclaration de leur classe. Dans notre exemple, si on veut compter le nombre des complexes à chaque instant on procède de la manière suivante : Dans Complex.h on ajoute :

```
static  int  nbComplexes; // compteur des objets complexes
static void printnbComplexes();// méthode statique qui affiche le
    compteur
```

Dans Complex.cpp on ajoute(en dehors des fonctions) :

```
int Complex::nbComplexes=0; // initialisation
```

Dans les différents constructeurs on incrémente nbComplexes nbComplexes++; Dans le destructeur, on le décrémente nbComplexes–;.

On ajoute la méthode d'affichage

```
static void Complex::printnbComplexes() {
    cout<<" Nombres de Complexes:"<<nbComplexes;
}
```

On effectue ensuite les tests nécessaires.

1.3.2 Encapsulation des données

- Principe fondamental en AOO,
- consiste à cacher les données membres (éventuellement quelques méthodes internes) et ne donner accès en lecture ou modification que par le biais de méthodes
- Pour des raisons de compatibilité, les langages O.O offrent plusieurs niveaux d'encapsulation.
- En C++ :
  - Un membre public est accessible partout dans le programme.
  - Un membre privé n'est accessible que par les méthode de sa classe.

— Un membre protégé est accessible dans sa classe et toutes les classes dérivées (par héritage).
— Par défaut, toute définition en dehors des qualificatifs (public, protected, private) est privée (private).

En C++, et pour une meilleure souplesse, la notion de fonctions amies d'une classe permet à une fonction, une méthode d'une classe ou à une classe entière d'accéder aux membres données d'une classe qui offre ce privilège. Déclaration :

```
friend <signature de fonction>
```

Une telle déclaration donne à la fonction tous les pouvoirs d'une méthode membre de cette classe. La fonction peut être une fonction ordinaire (n'appartenant pas à une classe) ou méthode d'une autre casse, dans ce cas il faut préfixer le nom de la méthode par le nom de la classe dans la signature.

```
    friend <typederetour> <nomdeclasse>::<nomdefonction><param>;
```

Lorsque toutes les méthodes d'une classe B sont des méthodes amies de la classe A on peut faire une seule déclaration d'amitié dans la classe A :

```
    friend class B;
```

Les accesseurs :

Pour utiliser des membres privées encapsulés d'un objet, on utilise les accesseurs :
— Un accesseur en lecture (getter)
— Un accesseur en écriture (setter)

Il est conseillé d'utiliser une forme canonique suivante pour les accesseurs : Supposons qu'on a un Attribut (abc) déclaré dans la classe avec : T abc ;
— getter : T getAbc() return abc ;
— Setter : void setAbc(T abc) this.abc=abc ;

Pour les attributs indexés (comme les tableaux), on utilise un indexe pour accéder chaque élément séparément. Par exemple si un attribut est déclaré T abc[] :
— Getter :

```
T getAbc(int index) {return abc[index];}
```

— Setter :

```
void setAbc(int index, T value) {abc[index]=value; // un contrôle
    des valeurs index et value est nécessaire.
```

R Il faut faire attention aux attributs dynamiques (getter peut retourner un pointeur vers une zone membre données qui devienne non protégée).

R Il n'est pas toujours nécessaire de mettre des accesseurs (cas des attributs internes qui évoluent par des méthodes autres que les accesseurs). Dans certains cas, on offre dans une classe que des getters pour des attributs.

Objets dynamiques et objets statiques

En C++, à l'instar des variables, les objets peuvent être statiques ou dynamiques :

```
Complex c1(2.0,3.0); // déclaration et appel à l'un des constructeur
c1.move(1.0,2.3);
Complex *c; // déclaration d'un objet dynamique
c= new Complex(2.0,3.0);  // création de l'objet déclaré précédemment
c->move(2.0,3.0);
```

1.4 Relations entres classes :

1.4.1 Héritage

L'héritage est aussi un concept fondamental en AOO, il permet de construire une classe à partir d'autres classes.

Intérêts :

— Réutilisation implicite d'une ou plusieurs classes.

— Extension naturelle des fonctionnalités d'une ou plusieurs classes.

Lorsqu'une classe hérite d'une seule classe on parle d'héritage simple, le cas échéant c'est l'héritage multiple.

L'héritage permet de définir une relation de type « est-un » (« is a ») entre l'objet est la superclasse de sa classe.

En C++, la relation d'héritage est réalisé dans la déclaration de la classe ; class A :class B .....; //signifie que la classe A hérite de la classe B.

Dans notre exemple, si on veut construire la classe des complexes avec une représentation graphique on peut définir une nouvelle classe (de la même manière que la première déclaration/implémentation)

Dans Complex.h on ajoute la déclaration de la nouvelle classe :

```
    class Gcomplex:Complex {
    private:
        float cx;
        float cy;
    Public:
        void  print();
        Gcomplex(float ,float ); //constructeur paramétré
        Gcomplex(Gcomplex &); // constructeur par recopie
        ~Gcomplex(); //destructeur
    };
```

Dans Compex.cpp on ajoute l'implémentation des méthodes de la classe Gcomplex :

```
    Gcomplex::Gcomplex(float rel,float img):Complex(rel,img) { // appel
    obligatoire au super-constructeur de Complex dans ce cas.
```

```
2           this->rel=cx=rel;
3           this ->img=cy=img;
4       }
5       void Gcomplex:: print(){ // même signature de print de Complex
6                                // On dit que la méthode print est redé
        finie dans la classe Gcomplex.
7           Cout <<" Complex:"<<rel<<" +i"<<img<<" :("<<cx<<" ,"<<cy<<" )";
8       };
```

1.4.2 Agrégation

Une autre manière d'ajouter des fonctionnalités dans une classes est l'agrégation. Elle consiste à mettre des objets appartenant à d'autres classes comme membres données dans la classe à enrichir. D'ailleurs cette relation est préférable à l'héritage dans plusieurs cas.

Dans notre exemple, on procède par la définition d'une nouvelle classe Point qui représente les coordonnées du complexe, puis, on ajoute les différentes méthodes nécessaires au bon fonctionnement (changement des coordonnées, affichage etc).

Dans la classe Complex, on ajoute un membre donnée :

```
1       Point *p; // ici c'est un objet dynamique pour pouvoir le créer à n
        'importe quel moment.
```

On veille à la cohérence entre le complexe et sa représentation graphique (le point) en utilisant les différentes fonctionnalités de la classe Point.

La classe Point :

Dans Complex.h et avant la déclaration de la classe Complex :

```
1   class Point {
2       private:
3           float x;
4           float y;
5       public :
6           Point(float,float);
7           Point(Point&); ~Point();
8           void setX(float);
9           float getX();
10          void setY(float);
11          float getY();
12          void print();
13  };
```

Ajout d'un membre données à la classe complex, toujours dans COmplex.h)

```
1   class Complex{
2       private:
3           float rel;
4           float img;
5           Point *p;
6           static int nbComplexes;
7       public:
```

```
8           void print();
9           Complex(float,float);
10          Complex(Complex &);
11          ~Complex();
12          void setRel(float);
13          float getRel();
14          void setImg(float);
15          float getImg();
16          static void printnbComplexes();
17  };
```

Implémentation des nouvelles fonctinalités dans complex.cpp :

```
1   #include <iostream>
2   #include "Complex.h"
3   using namespace std;
4   void Complex::print(){
5       cout<<rel<<" + "<<img<<"i, ";
6       (*p).print();
7       cout <<"\n";
8   }
9   Complex::Complex(float a,float b){
10      rel=a;
11      img=b;
12      nbComplexes++;
13      p=new Point(a,b);
14  }
15  Complex::Complex(Complex& clone){
16      cout<<"Clonage \n";
17      rel=clone.rel;
18      img=clone.img;
19      p=new Point(rel,img);
20      nbComplexes++;
21  }
22  int Complex::nbComplexes=0;
23  void Complex::setRel(float a){
24      rel=a;
25      p->setX(a);
26  }
27  float Complex::getRel(){
28      return rel;
29  }
30  void Complex::setImg(float b) {
31      img=b;
32      p->setY(b);
33  }
34  float Complex::getImg(){
35      return img;
36  }
37  void Complex::printnbComplexes(){
38      cout<<"Nombres de complexes existants:"<<nbComplexes;
39  }
40  Complex::~Complex(){
41      cout<< "Au revoir... \n"; nbComplexes--;
42  }
```

```
43 Point::Point(float a,float b){
44     x=a;
45     y=b;
46 }
47 Point::Point(Point &p){
48     x=p.x;
49     y=p.y;
50 }
51 float Point::getX() {
52     return x;
53 } void
54 Point::setX(float a){
55     x=a;
56 }
57 float Point::getY() {
58     return y;
59 } void Point::setY(float b){
60     y=b;
61 }
62 void Point::print() {
63     cout <<"Coordonnées:("<<x<<","<<y<<")";
64 }
```

Dans testComplex.cpp on peut tester par

```
1 \begin{lstlisting}
2 Complex c1(2.2,3.3);
3 Complex c2=c1;
4 c2.setRel(1.1);
5 c2.setImg(4.4);
6 c1.print();
7 c2.print();
```

1.5 Surcharge des opérateurs

La surcharge des opérateurs en C++ est une fonctionnalité très intéressante dans Le sens où on peut appliquer des opérateurs habituels (de C++) sur des objets. Quelques opérateurs possèdent une surcharge implicite (+(addition des pointeurs), *(adresse),...etc).

La notation opératoire (utilisant des opérateurs) est parfois plus concise que la notation fonctionnelle(appel de méthodes) , il est préférable par exemple pour deux objets de la classe Complex c1 et c2 d'écrire c3=c1+c2 que c3=add(c1,c2).

1.5.1 Exemple de surcharge d'opérateurs

Si on veut surcharger les opérateurs + et * dans notre classe Complex on peut procéder de deux manières :

1. La première est de considérer l'opérateur comme une fonction externe à la classe qui prend deux arguments complexes et retourne un complexe qui représente le résultat de l'expression(on la déclare comme fonction amie pour plus de flexibilité d'accès) :

```
c3=c1+c2;
```

sera équivalente à $c3 = \text{operator} +(c1, c2)$; prototypage (dans Complex.h) :

```
friend Complex operator+(Complex,Complex);
```

Implémentation dans Complex.cpp :

```
Complex operator +(Complex c1,Complex c2){
    Complex c3(0,0);
    c3.setRel(c1.getRel()+c2.getRel());
    c3.setImg(c1.getImg()+c2.getImg());
    return c3;
}
```

R Dans l'expression c3=c1+c2, il y a, en plus de la surcharge de l'addition, une affectation du résultat à c3 ce qui a l'effet d'invoquer le constructeur par copie. Hors ce constructeur possède un argument de type référence (Complex&) mais c1+c2 est une expression de type Complex (une valeur). Afin de contourner ce problème, Une des solutions consiste à rendre l'argument du constructeur par copie comme une référence constante équivalente à une valeur de point de vue typage. Dans notre exemple, l'entête du constructeur sera changée en (dans .h et également dans .cpp) :

```
Complex Complex(const Complex &)
```

Après on peut tester notre opérateur + par le code (dans testCOmplex.cpp) :

```
Complex c1(1.0,2.03)
Complex c2(3.0,4.0);
Complex c3=c1+c2;
c1.print();
c2.print();
c3.print();
```

2. La deuxième façon de surcharger un opérateur est par une méthode membre de la classe :
   Dans notre exemple, si on veut surcharger l'opérateur * dans la classe Complex on peut considérer que c3=c1*c2 équivalente à c1.operator *(c2). Dans Complex .h on ajoute la déclaration :

```
Complex operator *(Complex);
```

Dans Complex.cpp on ajoute l'implémentation de la méthode operator *

```
Complex Complex::operator *(Complex b){
Complex tmp(0,0);
tmp.setRel(rel*b.getRel()-img*b.getImg());
tmp.setImg(rel*b.getImg()+img*b.getRel());
return tmp;
}
```

Pour tester l'opérateur surchargé, on ajoute dans testComplex :

```
Complex c4=c1*c2;
c4.print;
```

Règles de la surcharge des opérateurs :

1. Se limiter aux opérateurs existants : on ne peut pas surcharger des symboles qui ne sont pas des opérateurs de C++, en plus certains opérateurs ne sont pas surchargeables (comme le point .) et d'autres sont surchargeables avec certaines contraintes,
2. Conserver la pluralité des opérateurs (unaires ou binaires) en nombres d'arguments,
3. Lorsque plusieurs opérateurs sont surchargés, ils gardent leurs règles de priorité et d'associativité dans les expressions, dans notre exemple, une expression c4=c1+c2*c3 :* sera évaluer en premier lieu, puis le résultat sera utilisé comme argument à +
4. Éviter les hypothèses sur les rôles des opérateurs : lorsqu'ils sont surchargés se sont des fonctions auxquelles on peut attribuer n'importe quel code.
5. Les opérateurs surchargeables sont : + - * / & | ! += -= *= /= ≙ &= |= ++ – = == != < > <= >= « » «= »= && || ->* , -> [] () new delete . Les cinq derniers sont : l'indirection, l'indexation, l'appel de fonction, l'allocation et la désallocation. . : : ? : sizeof .* & ne sont pas surchargeables.