Introduction aux classes: un exemple avec la classe Complexe

1 Introduction

En C++, il est possible de créer des "objets". Vous êtes habitués à manipuler des objets sans le savoir: des entiers, des réels, des booléens.... En général, l'utilité de ces objets est limitée puisqu'on ne peut réaliser que des opérations de base (+,-,/,...). A partir de maintenant, vous pouvez créer vos propres objets. Par exemple, un exemple bien utile pour remplacer les tableaux de caractères est l'objet `string`. Il permet de faire des opérations très facilement sur des chaines de caractères:

```
int main(){
string a; a="salut";
a=a+" toto";//a= "salut toto"
}
```

En C++ les objets ne sont limités que par votre imagination et vos besoins. Vous voulez des objets "Voiture", "Personnage", "Avion", "Interval", "complexe". Tout est possible.
Ici nous nous focaliserons sur l'exemple des "complexes". Le mot clef en C++ pour créer un objet est `class`.

```
class Complexe{


};
```

---

2 Données et fonctions membre

Un objet en C++ contient des "données membre" et des "fonctions membre". Les données membre sont les données qui caractérise l'objet. Les fonctions membre sont les fonctions que peut réaliser l'objet. Par exemple un nombre complexe pourrait avoir:

- En données membre: Une partie imaginaire, une partie réelle, un argument et un module.
- En fonctions membre: Il pourrait s'afficher grâce à la fonction "affiche", il pourrait "s'additionner" avec un autre complexe, effectuer une rotation,...

```
class Complexe{

//données membre
    float partie_reelle; // partie réelle
    float partie_im; // partie imaginaire
    float module;// module
    float argument;// argument

//Fonctions membres
    Complexe additionne(Complexe & Autre);// une addition entre un complexe et un autre renvoie un complexe
    void Rotation( float angle);// Ici la rotation s'applique à l'objet lui même donc il ne renvoie rien
    void affiche() const;// l'affichage ne prend aucun agrument et ne modifie rien, donc on ajoute le mot clef const

};
```

En général, les objets , par convention, commencent toujours par une majuscule.

Une fonction membre particulière: le constructeur

Avant même de parler de l'utilisation d'un objet, il faut penser à sa déclaration. En C++ avant d'utiliser un `int` vous êtes obligés de le déclarer, ou autrement dit de **construire l'objet**. En fait,

lorsqu'on déclare une variable, on fait implicitement appel à son **constructeur**.

La ligne `int i;` fait appel au constructeur des objets entier. Ici, c'est pareil. Si vous voulez déclarer un objet de type complexe, il faudra certainement utiliser une ligne du type :`Complexe c;`.

Il est possible pour tout type d'objet, grâce au principe de surcharge de créer plusieurs constructeurs. Le constructeur sans argument est appelé un **"constructeur vide"**. Par exemple, on pourrait penser dans le cas des complexes à créer un complexe directement en lui donnant en paramètres sa partie imaginaire et sa partie réelle:
`Complexe c(1,5);// créée le complexe 1+5i`

Ainsi, afin de pouvoir déclarer un objet, il doit exister au moins un constructeur au sein de la classe. Ce constructeur est une fonction membre, sans type, qui porte le même nom que la classe. En théorie, il n'est pas obligatoire d'avoir un constrcuteur vide. Cependant, il est obligatoire lorsqu'on veut pouvoir créer un tableau d'objets comme dans l'exemple ci dessus. **Ainsi, il est conseillé de toujours faire un constructeur vide (sans arguments).**

Enfin, en C++ il est possible de définir une définition du type: `int i=3;`**C'est ce qu'on appelle un constructeur par recopie.** Il est de même possible de faire avec n'importe quel type d'objets. Par exemple:
`Complexe e(3,4);// e=3+4i`
`Complexe d=e; // Constructeur par recopie : d=3+4i`
Si vous ne spécifiez pas de constructeur par recopie, le compilateur le fera automatiquement. ATTENTION: Si la classe contient des pointeurs en donnée membre, il est IMPERATIF de créer manuellement un constructeur par recopie (Nous reviendrons dessus plus tard). Faites bien attention à la déclaration du constructeur par recopie qui utilise un passage par référence.

Voici ce que ca donne pour la classe Complexe:

```
class Complexe{

//données membre
    float partie_reelle; // partie réelle
    float partie_im; // partie imaginaire
    float module;// module
    float argument;// argument

//Fonctions membres

    Complexe();// Constructeur vide
    Complexe(float Re,float Im); //Constructeur premettant directement d'attribuer partie Re et Im.
    Complexe( Complexe & Autre);// Constructeur par recopie

    Complexe additionne(Complexe & Autre);// une addition entre un complexe et un autre renvoie un complexe
    void Rotation( float angle);// Ici la rotation s'applique à l'objet lui même donc il ne renvoie rien
    void affiche() const;// l'affichage ne prend aucun agrument et ne modifie rien, donc on ajoute le mot clef const

};


int main(){

    Complexe c; //Appel auy constructeur vide
    Complexe d(3,4); //Appel a constrcteur avec paramètres
    Complexe * cptr; //Possibilité de définir des pointeurs de complexe
    cptr= new Complexe; //de les allouer
    Complexe cT[10]; //définir des tableaux: ATTENTION BESOIN D'UN CONSTRUCTEUR VIDE
    delete cptr;

    Complexe e=d;// Utilisation du constructeur par recopie
    Complexe f(d);// Deuxième notation possible pour le constructeur par recopie

    return(0);
}
```

3 Le principe d'encapsulation : Private et Public

Maintenant qu'on a vu comment définir un objet et comment le dééclarer, voici venu le temps de l'utiliser. On peut appeler toute fonction membre ou donnée membre comme on le fait pour des structures: simplement en utilisant un point "." lorsqu'il s'agit de l'objet et d'une fleche "->" lorsqu'il s'agit d'un pointeur. Par exemple:

```
int main(){

    Complexe c; //Appel auy constructeur vide
    Complexe d(3,4); //Appel a constrcteur avec paramètres
    Complexe * cptr= new Complexe; //de les allouer

    c.partie_reelle=3;
    c.partie_im=2;
    c.module=0;
    c.affiche();
    c=c.additionne(d);
    cptr->partie_im=2;
    delete cptr;
    return(0);
}
```

Si les classes étaient des structures, il n'y aurait aucun problème à faire ca. **Cependant, ici nous avons un problème.** Regardez le code ci-dessus, nous avons un complexe `c`, dont la partie réelle vaut 3, la partie imaginaire vaut 2, et dont le module 0. **Ceci n'est pas un complexe valide!**. C'est la différence fondamentale entre une structure et un objet. Les champs dans une structure sont indépendants les uns des autres. Pour les objets, ils ne le sont pas. Ils doivent garder une certaine forme d'intégrité. **C'est ce qu'on appelle le principe d'encapsulation:** Il y a des choses qu'on peut autoriser à l'utilisateur et d'autres qu'on ne peut pas autoriser, comme par exemple ici de changer la partie réelle sans impacter le module et l'argument. Cependant il ne faut pas que ce soit compliqué pour l'utilisateur. Ce n'est pas à lui de penser à modifier tous le champs nécéssaires.

Les choses qu'on autorise en dehors de la classe sont déclarées grâce au mot clef `public:`. Celles que l'on autorise pas sont déclarées grâce au mot-clef `private:`. Si on spécifie pas de mot clef, les membres seront *de facto* `private`. Ainsi, les codes précédents n'auraient en vérité pas fonctionné. J'ai simplement voulu éviter des complications non-nécessaires.

En matière de convention, le bon programmeur mettra **toujours** les données membre en `private`. Bien sûr les constructeurs doivent être en public pour pouvoir créer un objet depuis l'extérieur de la classe. Ainsi, l'exemple précédent devient:

```
class Complexe{
private:
//données membre
    float partie_reelle; // partie réelle
    float partie_im; // partie imaginaire
    float module;// module
    float argument;// argument

//Fonctions membres
public:
    Complexe();// Constructeur vide
    Complexe(float Re,float Im); //Constructeur premettant directement d'attribuer partie Re et Im.
    Complexe( Complexe & Autre);// Constructeur par recopie

    Complexe additionne(Complexe & Autre);// une addition entre un complexe et un autre renvoie un complexe
    void Rotation( float angle);// Ici la rotation s'applique à l'objet lui même donc il ne renvoie rien
    void affiche() const;// l'affichage ne prend aucun agrument et ne modifie rien, donc on ajoute le mot clef const

};


int main(){

    Complexe c; //Appel auy constructeur vide
    Complexe d(3,4); //Appel a constrcteur avec paramètres
    Complexe * cptr= new Complexe; //de les allouer

    c.partie_reelle=3;  //ERREUR car privé
    c.partie_im=2;      //ERREUR car privé
    c.module=0;         //ERREUR car privé
    c.affiche();
    c=c.additionne(d);
    cptr->partie_im=2;  //ERREUR car privé
    delete cptr;
    return(0);
}
```

Les "getters" et les "setters"

Suite à l'introduction de données et de fonctions privées, la question qui se pose immédiatement est : *"On ne peut donc plus accéder aux données membre? On ne peut donc plus les modifier? Ni les lire?"*. La réponse est non. On ne peut plus accéder à aucun élément privé. C'est pourquoi, lorsqu'on crée une classe, on créée des fonctions qui permettront de modifier les champs privés, **mais de la manière pensée par le concepteur, et en toute transparence pour l'utilisateur.**

En général, on créée des fonctions *"getters"* pour accéder aux champs privés en lecture, et des *"setters"* pour les modifier.
Par exemple, la ligne:
`cout«c.partie_reelle;`
deviendra:
`cout«c.get_partie_reelle();`
où la fonction `get_partie_reelle()` sera définie dans le domaine `public` de la classe Complexe.

De même, la ligne:
`c.partie_reelle=2;`
qui permettait de modifier la partie réelle (malheureusement sans modifier l'arguiment ni le module en conséquence) sera remplacée par:
`c.set_partie_reelle(2);`
où la fonction `set_partie_reelle()` sera également définie dans le domaine `public` de la classe Complexe. **Cependant, la différence ici est que le concepteur de la classe aura pensé, au sein de la fonction `set_partie_reelle()` à modifier en accord le module et l'argument du complexe.** Ainsi l'objet restera intègre et l'utilisateur ne pourra pas altérer le fonctionnement de l'objet.

```
class Complexe{
private:
//données membre
    float partie_reelle; // partie réelle
    float partie_im; // partie imaginaire
    float module;// module
    float argument;// argument

//Fonctions membres
public:
    Complexe();// Constructeur vide
    Complexe(float Re,float Im); //Constructeur premettant directement d'attribuer partie Re et Im.
    Complexe( Complexe & Autre);// Constructeur par recopie

    Complexe additionne(Complexe & Autre);// une addition entre un complexe et un autre renvoie un complexe
    void Rotation( float angle);// Ici la rotation s'applique à l'objet lui même donc il ne renvoie rien
    void affiche() const;// l'affichage ne prend aucun agrument et ne modifie rien, donc on ajoute le mot clef const

    //getters
    float get_partie_reelle() const; // permet d'afficher la partie reelle
    float get_partie_im() const; // permet d'afficher la partie imaginaire
    float get_module() const;
    float get_argument() const;

    //setters
    void set_partie_reelle(float newx); // permet de placer une nouvelle valeur dans la partie réelle
    void set_partie_im(float newy);//permet de placer une nouvelle valeur dans la partie imaginaire
    void set_module(float newm);
    void set_argument(float arg);
};


int main(){

    Complexe c; //Appel auy constructeur vide
    Complexe d(3,4); //Appel a constrcteur avec paramètres
    Complexe * cptr= new Complexe; //de les allouer

    c.set_partie_reelle(3);  //modifiera le module et l'argument
    c.set_partie_im(2);      //modifiera le module et l'argument
    c.set_module(0);         //modifiera la partie réelle et la partie imaginaire
    c.affiche();
    c=c.additionne(d);
    cptr->set_partie_im(2);  //modifiera le module et l'argument
    delete cptr;
    return(0);
}
```

La surcharge d'opérataeurs

En C++, il est possible de surcharger des fonctions. En particulier, il est possible de surcharger des fonctions comme l'égalité ou l'addition par exemple. Si vous vouliez faire une addition de complexe, il serait fastidieux de devoir ecrire `a.additione(b)` (comme je l'ai fait plus haut) à la place de `a+b`. Et bien, Il est possible de définir sa propre fonction `+` en surchargeant la fonction nommée `operator+`. De nombreuses fonctions peuvent être ainsi surchargées (`operator-`,`operator=`,`operator=+`,`operator[]`,`...`).

L'opérateur "=" est un peu particulier. Nous en reparlerons plus bas lorsque nous verrons les pointeurs et les classes. Le compilateur en créée un par défaut et lorsque vous tapez la commande `a=b;`, alors automatiquement, le programme copie toutes les données membres de `b` dans les données membre de `a`. Ainsi si une classe ne contient pas de pointeur il est souvent inutile de programmer une surcharge de l'opérateur =.
Voici un exemple de surcharge de loperateur + pour les complexes. Faites bien attetion à la nomenclature.
Dans le Header:
`Complexe operator+( Complexe & c);` Dans le main:
`Complexe a;`
`Complexe b;`
`Complexe c; c=b+a;// Appel à la fonction operator+ et à la fonction operator=`

4 Le Header et le cpp

Maintenant on connait dans les grandes lignes comment utiliser et créer un objet en C++. Cependant nous n'avons vu que la *déclaration* des objets et la définition de ses données membre et fonctions membre. Bien entendu, ces fonctions membres ont une déclaration, mais n'ont pour l'instant pas de corps, nous n'avons pas encore décrit leur fonctionnement. Souvent, la définition de la classe, appelée "Header" est séparée de sa description. Ici nous avons divisé notre classe Coomplexe en un Header "Complexe.h" et un cpp "Complexe.cpp".

Le fichier "cpp" doit inclure le Header. De plus, afin de pouvoir utiliser un objet de type Complexe dans un code, il doit également y avoir un `#include` du Header, et seulement du Header.

Il est important de comprendre également que le Header est la vitrine d'une classe. C'est le seul fichier qu'un utilisateur potentiel ira voir. Jamais un utilisateur n'ira regarder le cpp. C'est donc important de **bien commenter** votre code. De plus, il convient de donner des noms explicites aux fonctions (notamment pour les getters et setters).

En ce qui concerne le cpp, la définition d'une fonction se passe de la manière suivante: `Type Classe::Nom_Fonction()`.
Ceci signifie, je définis la fonction `Nom_Fonction` définie dans le Header de la classe `Classe` et de type `Type`.

Quelques détails supplémentaires

Lorsqu'on compile de gros projets, il se peut que plusieurs fichiers fassent un `include` du même Header, ou que des classes s'incluent mutuellement. Ceci n'est pas accepté par le compilateur et peut causer de gros problèmes de liens. Pour éviter cela, et je ne rentrerai pas dans les détails, le fichier Header commence en général par ces lignes:

```
#ifndef Nom_Classe_h
#define Nom_Classe_h
```

et se termine par:

```
#endif
```

Enfin, pour les plus avancés d'entre vous, **et seulement pour les constructeurs** il existe des raccourcis de notation. Par exemple, toutes les écritures suivantes sont équivalentes:

```
Complexe::Complexe()
{
partie_reelle=0;
partie_im=0;
module=0;
arg=0;
}

Complexe::Complexe():partie_reelle=0,partie_im=0,module=0,arg=0
{
}

Complexe::Complexe():partie_reelle(0),partie_im(0),module(0),arg(0)
{
}
```

5 Exemple: La classe Complexe et son utilisation

Vous pourrez noter que, afin de garder l'intégrité des objets complexes, nous avons créé des fonctions qui calculent le module et l'argument. Puisque l'tuilisateur n'en a pas besoin directement, nous avons choisi de les mettre en private. Vous constaterez également que j'ai mis le constructeur par recopie en commentaire. Dans ce cas précis, puisque la classe ne contient pas de pointeur, ce qui est codé en commentaire est exactement ce qu'aurait fait le compilateur automatiquement.

Le Header

N'oubliez pas le ; après l'accolade a la fin de la classe.

```
#ifndef Complexe_hpp //ATTENTION NECESSAIRE LORSQUE LA CLASSE EST INCLUSE DEPUIS PLUSIEURS FICHIERS
#define Complexe_hpp

#include <stdio.h>
#include <iostream>
using namespace std;


class Complexe {


private: // données membres toujours déclarées en private

//On ne doit pas pouvoir permettre à l'utilisateur de pouvoir changer la partie imaginaire, sans changer par exemple le
//module du complexe.
    float partie_reelle; // partie réelle
    float partie_im; // partie imaginaire
    float module;// module
    float argument;// argument


//Ici se trouvent les fonctions qu'on va permettre d'utiliser en dehors de la classe (dans le main par exemple)
//On les déclare en public
public:  // fonctions membres
    Complexe(); // constructeur vide // crée un complexe à 0 (le programmeur de classe doit choisir)
    Complexe(float x, float y); // constructeur 2// créer un complexe = x+iy
    //  Complexe(const Complexe &C); //Constructeur par copie. Si vou sn'en définissez pas comme c'est le cas ici, il est
        fait automatiquement par le compilateur. ATTENTION, il est recommandé de toujours définir son propre constructeur
        par copie lorsque la classe contient des pointeurs!

    //Ces fonctions sont souvent (toujours) présentes dans une classe lorsque les données sont privées
    float get_partie_reelle(); // permet d'afficher la partie reelle
    float get_partie_im(); // permet d'afficher la partie imaginaire
    void set_partie_reelle(float newx); // permet de placer une nouvelle valeur dans la partie réelle
    void set_partie_im(float newy);//permet de placer une nouvelle valeur dans la partie imaginaire


    Complexe operator+(Complexe& c);//Surcharge de l'opérateur plus.
    void Affiche (); //permet d'afficher le complexxe

//Ici se trouvent des fonctions qui n'ont pas besoin d'être utilisées en dehors de la classe
private :

    void Calcule_module (); //Calcule le module du complexe
    float Calcule_argument (); //Calcule l'argument du complexe

};
#endif /* Complexe_hpp */
```

Le cpp

Ici vous retrouverez la définition de l'oprérateur + entre autre.

```cpp
#include "Complexe.hpp" //Include du header
#include <math.h>


//Constructeurs
Complexe :: Complexe( ){
    partie_reelle =0;
    partie_im = 0;
    module=0;
    argument=0;
}

Complexe :: Complexe(float x,float y){
    partie_reelle =x;
    partie_im =y;
    Calcule_module();
    Calcule_argument();
}

/*
Complexe :: Complexe(const Complexe & C){
    partie_reelle =C.partie_reelle;
    partie_im =C.partie_im;
    module=C.module;
    argument=C.argument;
}
*/

//Ces fonctions privées ont été définies de deux manière différentes. L'une en void, l'autre en float. Ceci n'est pas très
    propre mais permet de donner un exemple pédagogique.
void Complexe:: Calcule_module(){
    module  = sqrt(partie_reelle*partie_reelle+partie_im*partie_im);
}

float Complexe:: Calcule_argument(){
    return atan(partie_im/partie_reelle);

}

//Remarquez ici, que le changement de la partie réelle ou imaginaire, impose le calcul de l'argument et du module. Ceci
    n'aurait pas été possible en laissant les données membre en public.
void Complexe::set_partie_reelle(float newx){
    partie_reelle= newx ;
    Calcule_module();
    argument=Calcule_argument();
}
void Complexe::set_partie_im(float newy){
    partie_im= newy ;
    Calcule_module();
    argument=Calcule_argument();
}

float Complexe::get_partie_im()
{
    return(partie_im);
}
float Complexe::get_partie_reelle()
{
    return(partie_reelle);
}


//Affichage
void Complexe::Affiche()
{
    cout<<"partie réelle: "<<partie_reelle<<endl;
    cout<<"partie imaginaire: "<<partie_im<<endl;
    cout<<" module: "<<module<<endl;
    cout<<" argument: "<<argument<<endl;
}


//Surcharge de l'opérator plus. On additionne les parties réelles et les parties imaginaires et on renvoie le résultat.
Complexe Complexe::operator+(Complexe& c)
{   Complexe res;
    res.partie_reelle=partie_reelle+c.partie_reelle;
    res.partie_im=partie_im+c.partie_im;
    res.Calcule_module();
    res.argument=res.Calcule_argument();
    return(res);
}
```

---

main

Un exemple de main avec son execution. Faites bien attention ici à l'appel des constructeurs et différentes fonctions. Vous voyez maintenant que le changements de modules et d'arguments des complxes sont complètement transparents pour l'utilisateur.

```cpp
#include <iostream>
#include "Complexe.hpp" //Inclusion du header
using namespace std;

int main(){
    float f=2.5;

    Complexe c; // appel au constructeur vide
    Complexe d(f,f); //appel au constructeur 2
    Complexe e(d); //Appel au constructeur par recopie automatiquement créé par le compilateur
    Complexe g=e; //Appel au constructeur par recopie automatiquement créé par le compilateur

    c.Affiche();
    d.Affiche();
    e.Affiche();
    g.Affiche();


    //l'appel des fonctions se fait comme pour des structures Objet.Fonction()
    cout<< c.get_partie_reelle(); // affiche la partie réelle de c (c.partie_reelle si la donnée était public).
    d.set_partie_reelle(3); // met la partie imaginaire de d à 3


    //il est possible de créer des pointeurs d'objets avec le constructeur voulu
    Complexe* cptr = new Complexe(f,f); //constructeur 2
    //Avec un pointeur on appelle une fonction en utilisant une -> et non un .
    cptr -> set_partie_reelle(5); // equivalent de l'appel (*cprt).set.partie_reelle(5)
    cptr->Affiche();

    //Tableaux d'objets: ATTENTION, dans ce cas, on fait toujours appel au constructeur vide
    Complexe T[10];//Il est possible de créer des tableaux de Complexe
    Complexe *Tf= new Complexe[10]; //Par allocation dynamique également
    T-> set_partie_reelle(3);
    T[4].set_partie_reelle(2);
    (Tf+3)->set_partie_reelle(2);


    c=d; //ici on peut faire appel à la fonction égalité sans l'avoir surchargée, car la classe ne contient pas de pointeur.
        Il serait très dangereux de le faire si elle en contenait

    e.set_partie_reelle(0);
    d=c+e; //appel à la fonction d'addition surchargée grâce à operator+


    T->Affiche();
    c.Affiche();
    e.Affiche();
    d.Affiche();


    delete [] Tf; //désallocation
    return 0;
}
```

Si vous executez le code voici ce que vous obtenez.

```
partie réelle: 0
partie imaginaire: 0
 module: 0
 argument: 0
partie réelle: 2.5
partie imaginaire: 2.5
 module: 3.53553
 argument: 0
partie réelle: 2.5
partie imaginaire: 2.5
 module: 3.53553
 argument: 0
partie réelle: 2.5
partie imaginaire: 2.5
 module: 3.53553
 argument: 0
0partie réelle: 5
partie imaginaire: 2.5
 module: 5.59017
 argument: 0.463648
partie réelle: 3
partie imaginaire: 0
 module: 3
 argument: 0
partie réelle: 3
partie imaginaire: 2.5
 module: 3.90512
 argument: 0.694738
partie réelle: 0
partie imaginaire: 2.5
 module: 2.5
 argument: 1.5708
partie réelle: 3
partie imaginaire: 5
 module: 5.83095
 argument: 1.03038
Program ended with exit code: 0
```

6 Les classes et les pointeurs

Jusqu'ici, nous n'avons parlé que de classes qui ne contiennent pas de pointeurs. Cependant, il n'est pas rare que (c'est même souvent le cas) les classes en contiennent en données membre. Ceci implique plusieurs choses. La première, comme dans tout problème contenant des pointeurs est lié aux problèmes d'allocation.

Le destructeur

Si l'on considère qu'une donnée membre est un pointeur, alors il faudra certainement l'allouer au moment où l'on créée l'objet. Dans une classe contenant des pointeurs, les constructeurs contiennent souvent (ce n'est pas toujours vrai pour le constructeur vide) un `new`. Ceci implique que quelque part, dans le code, il doit y avoir un `delete`. Ce delete a souvent lieu lors de la destruction d'un objet. C'est pourquoi il faut un `destructeur`. Le destructeur est appelé à chaque destruction d'un objet. Il se note avec un tilde suivi du nom de la classe. Il n'a pas de type non plus.

Voici un exemple très simple, de classe contenant des pointeurs. Voyez comment est utilisé le destructeur. Retenez simplement que **toute classe contenant un pointeur, doit contenir un destructeur.**

```
class TableauEntier{
private:

    int *tab;
    int taille;
public:

    TableauEntier();//constructeur vide qui ne fait aucune taille
    TableauEntier(int unetaille);//créer un tableau d'entier avec une taille donnée

    ~TableauEntier();//destructeur

};

TableauEntier::TableauEntier()
{
    taille=0;
    tab=0;//ici je n'alloue pas de tableau. J'aurais pu écrire tab=NULL aussi;

}
TableauEntier::TableauEntier(int unetaille)
{
    taille=unetaille;
    tab=new int[unetaille];// allocation dynamique
}

TableauEntier::~TableauEntier()
{
    if (tab!=0) // on verifie que le pointeur a été alloué
    {delete [] tab;}// désallocation
}

int main(){

    TableauEntier t(5);// ici on allouera le pointeur
    TableauEntier *tptr;//pointeur de tableaux
    tptr=new TableauEntier(7);// allocation de tptr


    delete tptr;// desallocation de tptr qui appelera automatiquement le destructeur de TableauEntier
    return(0);
} //a la fni du programme tous les destructeurs sont appelés
```

Le constructeur par recopie et l'opérateur =

Comme dit précédemment, lorsque vous ne spécifiez pas de coonstructeur par recopie ou de surchrage d'opérateur =, le compilateur en crééera un automatiquement. Cependant, comme dit, il le fait de manière basique en recopiant une à une les données membres. Lorsqu'il n'y a pas de pointeur, pas de problème. Cependant, lorsqu'il y a des pointeurs, ce n'est plus suffisant et ca sera la source de nombreuses erreurs. Pourquoi?

Reprenons l'exemple précédant des `TableauEntier`:
TableauEntier t(6);
TableauEntier t3(5);
TableauEntier t2=t;//Appel au constructeur par recopie
t3=t;// appel à opérateur+

Comme dit précédemment, si on ne spécifie ni l'opérateur =, ni le constructeur par recopie, par défaut, chaque champs sera copié dans l'autre. Ainsi implicitement, ce qui sera opéré sera : `t2.tab=t.tab` et `t3.tab=t.tab`. Ainsi dès qu'on modifiera la valeur `*(t.tab)`, la valeur de `*(t2.tab) et *(t3.tab)` seront modifées également, et cela n'est pas souhaitable.
Ce qui est à retenir, c'est que dans le cas où une classe contient des pointeurs, il FAUT redéfinir un opérateur = et un constructeur par recopie. En ce qui concerne le constructeur par recopie, c'est relativement facile.

```
class TableauEntier{
private:

    int *tab;
    int taille;
public:

    TableauEntier();//constructeur vide qui ne fait aucune taille
    TableauEntier(int unetaille);//créer un tableau d'entier avec une taille donnée
     TableauEntier(TableauEntier & autre);//constructeur par recopie
    ~TableauEntier();//destructeur

};

TableauEntier::TableauEntier(TableauEntier & autre)
{
    taille=autre.taille;//meme private on est ici au sein de la classe, donc pas de problème
    // Le constructeur par recopie du compilateur par défaut effectuerait la ligne
    // tab=autre.tab; et ce n'est pas bon
    //il faut réallouer un tableau et copier valeur par valeur
    tab= new int[taille];//on alloue le tableau
    for(int i=0;i<taille;i++)//on recopie élément par élément le tableau
    {
        *(tab+i)=*(autre.tab+i);
    }

}
```

En ce qui concerne l'opérateur =, c'est un peu plus complexe. Lorsqu'on écrit `a=b`, qu'on peut lire comme `a.operator=(b)` doit stocker, contrairement à l'habitude, le résultat dans `a` lui-même. Ne rentrons pas dans les détails, mais il faut renvoyer l'objet dans lui-même. Donc pour ne pas vous brouiller, l'esprit, vous pouvez apprendre cette nomenclature par coeur.

```
class TableauEntier{
private:

    int *tab;
    int taille;
public:

    TableauEntier();//constructeur vide qui ne fait aucune taille
    TableauEntier(int unetaille);//créer un tableau d'entier avec une taille donnée
     TableauEntier(TableauEntier & autre);//constructeur par recopie
    ~TableauEntier();//destructeur
    TableauEntier& operator=(TableauEntier& autre);  //On renvoie un objet de la classe en passage par référence
};

TableauEntier& TableauEntier::operator=(TableauEntier& autre)
{
    taille=autre.taille;
    tab= new int[taille];//on alloue le tableau
    for(int i=0;i<taille;i++)//on recopie élément par élément le tableau
    {
        *(tab+i)=*(autre.tab+i);
    }
  return *this;//On retourne l'objet lui même où this est un pointeur sur l'objet lui même
}
```

Conclusion

Voila, nous en avons terminé avec le cours sur les classes, voici un bref résumé de ce qu'il y a à retenir. En C++, une classe permet de créer des objets d'un type qui n'existe pas et qui pourrait être utile dans certaines applications. Ici nous avons pris l'exemple de la classe Complexe. Les notions importantes à comprendre sont:

- La notion d'encapsulation: Ceci différencie les classes des structures. Tous les champs (qu'il s'agisse de données membres, ou de fonctions membres) peuvent être rendus accessibles (public:) ou inaccessibles (private:) à l'extérieur de la classe. En général, toutes les données membres sont déclarées en private.

- La notion de constructeur: Afin de pouvoir déclarer un objet dans le main, il doit exister au moins un constructeur au sein de la classe. Ce constructeur est une fonction, sans type, qui

porte le même nom que la classe. Il est conseillé de toujours faire un constructeur vide (sans arguments). Il est appelé dans le main comme une déclaration:

```
Complexe e;
Complexe e();
Complexe e(d);
Complexe e=d;
```

Ici les deux premières lignes font appel au constructeur vide, les deux dernières font appel au constructeur par recopie (attention, si l'utilisateur n'en spécifie pas un, le compilateur en fera un de lui même!).

- Les classes et les pointeurs: Il faut bien retenir ici que lorsqu'une classe contient des pointeurs en données membre, cela ajoute une légère complexité à la notion de classe. En effet puisque la déclaration d'un objet appelera le constructeur qui contiendra certainement un new, il faudra bien faire attention d'inclure quelque part un delete. Ceci se fait dans une fonction qu'on appelle le destructeur.

- destructeur: Le destructeur est une fonction qui, comme le constructeur n'a pas de type. Elle porte le même nom que la classe avec un tilde devant (`Complexe();`). C'est la que se passe les désallocations de pointeurs lorsque la classe en contient. On peut retenir cette règle : En général (il y a quelques exceptions), lorsque la classe ne contient pas de pointeurs en données membres, il n'y a pas besoin de destructeur.

- Les classes et les pointeurs II: En général, les constructeurs par recopie et les assignations a=c sont bien gérés par le compilateur lorsque la classe ne contient pas de pointeur en données membres. Lorsqu'elle en contient, alors il est conseillé à l'utilisateur de définir ces fonctions lui-même.

- Le code d'une classe est souvent divisé en un fichier ".h" (appelé Header) et un fichier ".cpp": Le fichier cpp doit inclure le Header. Le main doit contenir également le Header (pas forcément le cpp). Le Header est "la vitrine" de votre classe. Il convient alors pour un utilisateur de bien commenter ses fonctions. Le cpp sert à la définition des fonctions.

- La surcharge d'opérateurs: En C++, il est possible de surcharger des fonctions. En particulier, il est possible de surcharger des fonctions comme l'égalité ou l'addition par exemple. Si vous vouliez faire une addition de complexe, il serait fastidieux de devoir ecrire a.Plus(b) à la place de a+b. Il est possible de définir sa propre fonction + en surchargeant la fonction nommée operator+. De nombreuses fonctions peuvent être ainsi surchargées (operator-,operator=,operator=+,operator[],...)

- Le passage par référence: Lorsqu'on crée des fonctions ayant pour attribut un objet, on le passe le plus possible par référence. Cela évite la recopie interne de l'objet qui parfois peut être volumineux.