Cours de programmation orientée objet

1. Les pointeurs

Un pointeurs sert à stocker une adresse par int* ptr_i = &i (ce qui est l'adresse de i). Pour accéder à la valeur de i, on doit passer par l'opérateur de déréférencement *ptr_i (ce qui nous donne accès à la variable i en elle-même).

A. Les tableaux

Un tableau est un pointeur constant sur la première case (int tab[] = int *tab)
Dans le cas où j'ai un pointeur sur un tableau : int* p;
int tab[0];
p = tab;
p[0] accède à la première case d'un tableau, on peut aussi noté *(p);

B. L'allocation mémoire

Allocation statique (sur la pile) -> désavoué dès que l'on sort du bloc (très peu utilisé)
Allocation dynamique (tas, mémoire persistante) -> on alloue dynamique avec new, et on désavoue avec delete. (à chaque new, un delete)

2. Les classes

A chaque classe un .h pour sa définition et un .cc avec son implémentation

-> C'est comme une structure
-> Point p; //p est une instance de la classe p, un objet

Point	Nom de la classe en Majuscule
my_abs; my_ord;	Attribut de la classe Toujours my_attribut
Point(); Point(float x, float y); ~Point(); affichePoint() : void;	Ce sont les fonctions membres (Méthodes). Elles sont écrites en lowerCamlCase. A chaque classe, -un constructeur par défaut, celui qui sera appelé quand on créera un objet sans valeur. -Un constructeur paramètre, le point utilisera ces valeurs -Un destructeur qui servira a désavouée tout ce qui a été allouée dynamiquement

A. La visibilité

Chaque attribut ou méthode admet des visibilités :
+ -> public : accessible par tous
- -> private : accessible seulement par la classe elle même
-> protected : accessible seulement par la classe et celle qui en hérite

Le constructeur est appelé implicitement quand on crée un objet : Point p2(5, 10);
Le destructeur est appelé implicitement à chaque fin d'utilisation (sortie de bloc)

B. Appel des fonctions :

Point p; p.affiche(); Quand c'est alloué dans la pile.
Point* p2 = new Point; p2->affiche. Quand c'est alloué dynamiquement dans le tas.

C. Implémentation des fonctions :

```
Point::affiche() {
 cout << point << endl
}
```

D. Les guetteurs

Ils permettent de connaître la valeur des attributs des objets en cours.

```
float Point::GetX() {
        return my_abs;
}
```

E. Les setters

Ils permettent de modifier les données des attributs des objets en cours.

```
void
Point::SetX(float X) {
        my_abs = x;
}
```

F. Les différents passage de paramètre

Le passage par copie -> La fonction travaille sur une copie de l'argument d'appel (pas top, bouffe de la mémoire) **Point P1**

Le passage par copie d'adresse -> La fonction travaille sur une copie de l'adresse de l'argument d'appel (on peut modifié la cible) **Point* P1**

Le passage par référence -> Rapide mais on peut modifier l'argument **Point &P1**

Le passage par référence constante -> Rapide et on ne peut pas modifier l'argument : **const Point &P1**

Les méthodes constantes: ce sont des méthodes qui ne pourront pas modifier l'objet qui appelle la fonction, attention à bien respecter la philosophie du langage.

void Point::Point (int param) const { }

G. Le constructeur par copie

Lorsque l'on appelle une fonction qui prend en paramètre par copie un objet, on fait une copie de l'argument d'appel (elle est réalisé par le constructeur par copie)

Il est défini implicitement, il fait une copie des attributs membres à membres, mais si il y a une allocation dynamique, il y a une possibilité d'effet de bord (modification de l'argument), on peut également pointé sur une case vide si elle a était détruite, on parle de mémoire partagé.

Dès que l'on alloue dynamiquement des attributs, on sera obligé de définir explicitement le constructeur par copie

Point::Point(const Point &autre), c'est à nous de définir ce que c'est qu'une copie, on peut copier ce que l'on veut.

H. La surcharge des opérateurs

L'opérateur d'affectation = :

Permet de simplifier des écritures Point P2 = P1;

Pour une classe Point, son prototype

.h

```
Point& operator= (const Point &autre);
```

.cc

```
Point&
Point::operator=(const Point &autre) {
	if ( this != &autre ) {  //this est un pointeur sur l'objet courant
		my_abs = autre.my_abs;
		my_ord = autre.my_ord;
	}
	return *this;
}
```

Attention, dans ce cas là P1(1,1) est différent de P2(1,1).

L'opérateur d'égalité == :

L'implémentation implicite fait une comparaison membre à membre ce qui souvent ne répond à ce que l'on veut.

```
bool
Point::operator==( const Point &autre ) const {
	if ( this == &autre )
		return true;
	return ( my_abs == autre.my_abs && my_ord == autre.my_ord );
}
```

I. Surcharge de fonction non-membre

Sans modifier .h et .cc de la classe Complexe.

Z1 = Z2 / Z3; //Si l'on veut faire cela

L'opérateur de division /

```
Complexe operator/(const Complexe &Z1, const Complexe &Z2) {
        return z1 * z2.inverse();
}
```

J. L'opérateur de redirection de flux (<< et >> ostream)

Si l'on veut faire cout << Z1;

.h

```
ostream& operator( ostream &out, const Complexe &z ) {
        out << z.toString << endl;
        return out;
}
```

Il faut donc d'abord penser à faire la méthode toString de la classe Complexe :

```
Complexe::toString() const {
        ostringstream ostr
        ostr << my_reel << «/« << my_img ;
        return ostr.str();
}
```

Ce qui peut être intéressant dans le projet de faire cout << MaGrille;