C++ 2A
TD/TP 2

1 Constructeurs et destructeurs

1. Quels sont les différents constructeurs disponibles en C++ ?
2. Écrire une classe Test contenant uniquement ces constructeurs et un destructeur.
3. ```
   Test t0;       // Quel constructeur est appelé ?
   Test t1( t0 ); // -
   Test t2( 5 );  // -
   auto t2 = t0;  // -
   ```

Dans le code suivant, quelles lignes sont correctes et lesquelles produisent une erreur ou sont incorrectes ?

```
class A {
  A() {}
};

class B {
public:
  B( int a ) {}
}

int main()
{
  A a0;
  A a1();
  B b0;
  B b1( 3 );
  B b2( b1 );
  auto b3( b1 );
}
```

2 Surcharge d'opérateurs

Certains opérateurs se surchargent uniquement dans la classe, d'autres à l'extérieur de la classe(¡¡), d'autres peuvent l'être indifféremment (+). Une liste des opérateurs et de la manière de les définir est disponible ici : C++ operators.

2.1 operator+

Déclaration dans la classe : l'opérateur + est une méthode et ne prend donc qu'un argument. Au lieu d'écrire c = a + b, on peut aussi écrire c = a.operator+(b) ; :

```
class A {
  public:
  A operator+( A const & a ) {
    ...
  }
};
```

Déclaration à l'extérieur de la classe : l'opérateur dans ce cas est une fonction qui prend 2 arguments :

```
A operator+( A const & a, A const & b ) { ... }
```

Il est possible de surcharger plusieurs fois l'opérateur + avec des arguments de types différents. Le type de retour peut également être différent. Par exemple, si on définit 2 classes `B` et `C`, on peut définir un opérateur avec 2 arguments de type `A` et `B` retournant une valeur de type `C` :

```
C operator+( A const & a, B const & b ) { ... }
```
```

2.2 operator=

Il se définit obligatoirement dans la classe :

```
class A {
public:
  A & operator=( A const & a ) {
    ...
    return *this;
  }
}
```

Attention, l'opérateur = retourne une référence de l'instance elle-même et pas void. Cela permet d'écrire : a = b = c ; et permet d'éviter une copie.

2.3 operator[]

L'opérateur [] est généralement surchargé pour les classes de type conteneur, afin d'obtenir une syntaxe similaire à celle des tableaux.

```
class A {
  int array[ 3 ];
public:
  ...
  int operator[]( std::size_t i ) const { return array[ i ]; }
};
```

Dans le cas ci-dessus, l'opérateur [] retourne une valeur d'un tableau par copie et n'affecte donc pas l'instance dont le contenu reste inchangé. La méthode doit donc être déclarée const (ce n'est pas obligatoire mais cela renforce les vérifications effectuées par le compilateur et évite les effets de bord). On peut donc écrire :

```
A a(...);
...
auto x = a[ 2 ];
```

mais on ne peut écrire :

```
a[ 2 ] = 3;
```

car a[2] retourne une valeur et pas une référence, il faut donc définir une seconde surcharge de l'opérateur :

```
class A {
  ...
  int & operator[]( std::size_t i ) { return array[ i ]; }
};
```

Dans ce cas, la méthode ne peut être définie const car l'instance est modifiée.

2.4 operator<<

L'opérateur << est utilisé par défaut sur les types entiers pour effectuer des décalages de bits.

```
unsigned int a = 2;
a = a << 1; // décalage d'un bit vers la gauche.
// équivalent à une multiplication par 2 sur les types non signés.
std::cout << a << std::endl; // affiche 4.
```

Pour les flux (affichage, fichiers, réseau, ...), l'opérateur est utilisé pour injecter des données :

```
std::cout << 5 << std::endl;
```

L'opérateur se surcharge à l'extérieur de la classe :

```
std::ostream & operator<<( std::ostream & out, A & const a ) {
  out << a[ 0 ] << ",␣" << a[ 1 ] << ",␣" << a[ 2 ];
  return out;
}
```

3 Mise en pratique

3.1 Vecteurs

Créer une classe vec3f (vec3f.hpp + vec3f.cpp)contenant un tableau de 3 valeurs flottantes. Ajouter les constructeurs, opérateurs, accesseurs et méthodes nécessaires pour pouvoir écrire le code suivant (main.cpp) :

```
int main() {
  Vec3f v0( 1.0f, 1.0f, 1.0f );
  Vec3f v1( 1.0f, 0.0f, 0.0f );
  v1.x() += 3.0f;
  v0[ 1 ] -= 1.0f;
  float l = v0.length();
  v1 *= l;
  auto v2 = v0 + v1;
  auto v3 = v0 - v1;
  if( v2 < v3 )
  {
    v2 += v3;
  }
  std::cout << v2 << std::endl;
  std::cout << v3 << std::endl;
  return 0;
}
```

3.2 Nombres complexes

Procéder de même pour créer une classe Complex permettant d'écrire le code suivant :

```
int main() {
  Complex c0{ 1.0f, 2.0f };
  Complex c1( -1.0f, 1.0f );
  c0.im =2.0f;
  auto c2 = c0 + c1;
  auto c3 = c0 * c1; // Attention à la mult. des nombres complexes.
  auto c4 = c0 - c1;
  c2.re += 3.0f;
  Complex c5( c2 );
  std::cout << c0 << std::endl;
  std::cout << c1 << std::endl;
  std::cout << c2 << std::endl;
  std::cout << c3 << std::endl;
  std::cout << c4 << std::endl;
  std::cout << c5 << std::endl;
}
```