Nous savons que le C++ autorise la surdéfinition des fonctions, qu'il s'agisse de fonctions membres (méthodes) ou de fonctions indépendantes. Rappelons que cette technique consiste à attribuer le même nom à des fonction différentes ; lors d'un appel, le choix de la bonne fonction est effectué par le compilateur, suivant le nombre et le type des arguments.

Mais C++ permet également, dans certaines conditions, de surdéfinir des opérateurs. En fait, le langage C++, comme beaucoup d'autres, réalise déjà la surdéfinition de certains opérateurs, comme par exemple le « + » qui est capable de faire une addition sur deux nombres entiers mais également sur deux nombres réels.

En C++, vous pouvez surdéfinir n'importe quel opérateur existant, pour peu qu'il porte, au moins, sur un paramètre qui dispose d'un type défini par l'utilisateur. En effet, le compilateur interdit de changer le comportement des types prédéfinis (primitifs). A ce sujet, il est nécessaire que la surdéfinition proposée aille de soi. Il serait, par exemple, invraisemblable d'effectuer une multiplication sur une définition d'addition de deux nombres complexes (bien que cela soit tout à fait possible).

Un opérateur est finalement une fonction

Bien qu'il soit plus agréable d'écrire directement les symboles des opérations, le compilateur, lui, transforme l'opérateur par sa fonction équivalente et qui utilise systématiquement le préfixe « operator ». Lorsque nous écrivons :

z = x + y ; le compilateur le transforme avec la fonction « operator+ » comme suit : z = operator+ (x, y) ;

Du coup : z = a + b + c est remplacé par z = operator+ (operator+ (a, b), c) ;

Surdéfinition de l'opérateur « + » pour la classe « Complexe »

Comme la classe « Complexe » représente un objet mathématique, il est tout à fait envisageable de définir des opérations classiques en utilisant les symboles idoines plutôt que des méthodes du style, 'somme', 'produit', etc. Nous allons nous intéresser uniquement sur l'addition. Une fois que le mécanisme sera bien maîtrisé, il sera ensuite facile d'appliquer cette démarche sur les autres opérateurs. Dans ce contexte, nous allons appréhender plusieurs cas de figure et mettre en œuvre toutes les possibilités. Il ne faut pas perdre de vue, que pour l'utilisateur de la classe « Complexe », la syntaxe doit être la plus simple et la plus normale possible tel que :

Soit x, y, z et w des objets de type « Complexe », il faut pouvoir utiliser la syntaxe suivante : z = x + y + w ;

Il faudra également permettre des additions entre des nombres complexes et des nombres réels puisque d'un point de vue mathématique, les nombres complexes incluent les réels. C'est d'ailleurs après la mise en œuvre de ces différents critères que nous déterminerons la meilleure solution.

Opérateur « + » non membre avec des attributs publics :

L'opération « + » attend deux arguments pour effectuer la somme et le résultat de cette opération doit être du même type que les arguments.

z = x + y ; (compilateur) ➔ z = operator+(x, y) ;

Nous utilisons des références pour les paramètres, ce qui permet d'avoir un gain de temps appréciable. Par ailleurs, nous les décrivons constants pour annoncer à l'utilisateur qu'il s'agit d'une lecture uniquement.

Le retour doit également être un « Complexe ». Par contre, cette fois-ci, il ne faut surtout pas proposer une référence mais plutôt solliciter une copie. En effet, l'objet « c » à l'intérieur de la fonction va disparaître à la fin de cette dernière. Sa durée de vie est limitée à la fonction et donc nous ne pouvons pas faire référence à un objet qui ne va plus exister.

```
//-----------------------------------------------------------
struct Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
  void decalage(int dx, int dy);
};
//-----------------------------------------------------------
inline
Complexe operator+(const Complexe &c1, const Complexe &c2)
{
  Complexe c;
  c.reel = c1.reel + c2.reel;
  c.imaginaire = c1.imaginaire + c2.imaginaire;
  return c;
}
```

Les attributs sont publics. Ils sont accessibles de l'extérieur

Fonction opérateur "+"

Deux paramètres, "c1" et "c2"

Résultat : retour d'un "Complexe"

Opérateur « + » non membre avec des attributs privés et des méthodes de lecture associées :

Le fait d'utiliser une structure nous a permis d'avoir une écriture simplifiée sur la fonction relative à l'opérateur « + ». Toutefois, cette solution n'est pas envisageable dans la philosophie de la programmation objet. En effet, les attributs doivent impérativement être protégés de l'extérieur pour éviter tout erreur venant de l'utilisateur. Il faut donc prévoir des méthodes sûres qui retourneront simplement la valeur respective de ces attributs.

```
//-----------------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
  double partieReelle() const      { return reel; }
  double partieImaginaire() const { return imaginaire; }
  void decalage(int dx, int dy);
};
//-----------------------------------------------------------
inline
Complexe operator+(const Complexe &c1, const Complexe &c2)
{
  return Complexe(c1.partieReelle() + c2.partieReelle(),
                  c1.partieImaginaire() + c2.partieImaginaire());
}
//-----------------------------------------------------------
```

Les attributs sont de nouveau privés

Méthodes de lecture pour les deux attributs

Objet anonyme

Utilisation des méthodes de lecture

Opérateur « + » non membre avec relation d'amitié :

Si nous disposons déjà de méthodes de lecture des attributs de la classe, la solution précédente est tout à fait adaptée. Dans le cas contraire, nous pourrions envisagé de donner une autorisation spéciale à cette opération (et uniquement pour elle). Dans ce cas là, les attributs sont exceptionnellement accessibles. Cela n'est possible que par une relation d'amitié. Il faut alors indiquer que la fonction « + » est une amie de la classe « Complexe », et cette relation est qualifiée à l'intérieur de cette dernière grâce au mot réservé « friend ». Pour finir, cette déclaration d'amitié ne peut apparaître que dans la définition de la classe.

Cette déclaration peut être effectuée n'importe où. Elle n'est pas du tout influencée par le caractère privée ou public puisqu'il ne s'agit pas d'une méthode de la classe (d'un membre de la classe). Cela reste une fonction externe.
«

```
//-----------------------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
  friend Complexe operator+(const Complexe&, const Complexe&);
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
  void decalage(int dx, int dy);
};
//-----------------------------------------------------------------
inline Complexe operator+(const Complexe &c1, const Complexe &c2)
{
  return Complexe(c1.reel + c2.reel, c1.imaginaire + c2.imaginaire);
}
//-----------------------------------------------------------------
```

L'opérateur "+" est un ami de la classe et, à ce titre, est autorisé à utiliser directement les attributs

Nous pouvons de nouveau atteindre directement les attributs

Opérateur « + » membre :

Nous pouvons même envisager que l'opérateur « + » fasse partie de la classe au même titre, d'ailleurs, que l'opérateur d'affectation. Nous allons donc analyser ce cas de figure, mais attention, la signature doit être différente puisque, cette fois-ci, il s'agit d'une méthode de la classe et non plus d'une fonction. En effet, lorsque nous utilisons une méthode, nous devons toujours la qualifier par rapport à l'objet de la classe. Pour cet opérateur, il faudra penser de la même façon, même si l'écriture ne le laisse pas apparaître. C'est le compilateur qui effectuera la transformation nécessaire. Nous aurons finalement un paramètre de moins puisque le premier opérande est représenté par l'objet qui sollicite l'opération.

z = x + y; le compilateur le transforme avec la méthode« operator+ » comme suit : z = x.operator+ (y) ;

Même apparence

C'est une méthode de l'objet complexe « x ».

Du coup : z = a + b + c est remplacé par z = a.operator+(b).operator+(c) ;

Objet temporaire anonyme.

```
//-----------------------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
  void decalage(int dx, int dy);
  Complexe operator+(const Complexe &c);
};
//-----------------------------------------------------------------
inline Complexe Complexe::operator+(const Complexe &c)
{
  return Complexe(reel + c.reel, imaginaire + c.imaginaire);
}
//-----------------------------------------------------------------
```

Opérateur membre

Attention, au moment de la définition, n'oubliez pas de spécifier le nom de la classe.

Les conversions implicites

Nous avons effectué jusqu'à présent que des opérations sur des nombres complexes. Que se passe t-il si nous mélangeons des nombres complexes avec des nombres réels. Rappelons, qu'en mathématique, ce genre d'opération est réalisable puisque les nombres réels font partis de l'ensemble de définition des nombres complexes.

Dans le langage C++, nous savons que des conversions implicites existent. Ainsi, lorsque nous effectuons un calcul entre un réel et un entier, le nombre entier est automatiquement converti en un nombre réel pour permettre le traitement du calcul avec deux éléments de même type (le traitement ne peut se faire qu'avec des éléments de même type). Est-ce que cette conversion implicite existe également pour toutes les classes (définies par l'utilisateur) ?

Reprenons l'exemple de la classe « Complexe » et analysons son comportement face à la création et à l'affectation avec un nombre réel ou même avec un nombre entier :

Complexe a(5.3), b(3) ;

« a » est un nombre complexe dont la partie imaginaire est nulle, on passe donc d'une valeur réelle vers un nombre complexe. Dans cette écriture, nous faisons un appel '**explicite**' au constructeur. Pour « b », même remarque, avec toutefois une conversion intermédiaire implicite pour passer d'un nombre entier vers un nombre réel.

Complexe a = 5.3, b = 3 ;

Avec cette deuxième écriture, nous réalisons la même opération sauf qu'il s'agit cette fois-ci d'un appel '**implicite**' au constructeur. A ce sujet, nous voyons mieux le passage d'un nombre réel vers un nombre complexe.

Règle

L'ensemble des constructeurs d'une classe prenant « ***un seul paramètre*** », définit un ensemble de conversions implicites des types de paramètres fourni à chaque constructeur vers la classe englobante. Dans le scénario proposé ci-dessus, nous avons :

a (Complexe) ← 5.3 (double)
b (Complexe) ← 3.0 (double) ← 3 (int)

Analysons maintenant l'affectation en prenant le scénario suivant :

```
double  d ;
d = 5 ;
```

« d » est une variable réelle alors que « 5 » est une constante littérale entière. Le compilateur accepte cette écriture malgré la divergence des types. Pour que l'opération s'effectue, il est nécessaire d'avoir des types identiques. Le compilateur propose donc la conversion implicite de la valeur entière vers l'équivalent réel, à savoir, « 5.0 ».

Avons nous la même aptitude avec la classe « Complexe » ? Par exemple, pouvons nous proposer une affectation d'un réel vers un « Complexe » ?

Avant de répondre à cette question, il faut déjà savoir si l'affectation est possible entre des nombres « Complexe ». La réponse est oui puisque, grâce à la forme canonique, un certain nombre de méthodes sont automatiquement intégrées, avec notamment l'opérateur d'affectation. Comme pour toutes opérations, il est nécessaire d'avoir des types identiques pour réaliser le traitement. L'opérateur « = » attend donc un « Complexe » comme argument.

```
//-----------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
  void decalage(int dx, int dy);
  Complexe operator+ (const Complexe &c);
  Complexe& operator= (const Complexe &c);
};
//-----------------------------------------
```

Opérateur d'affectation défini implicitement

Revenons à notre problème initial. A priori, il n'existe pas d'affectation directe entre un nombre réel et un nombre « Complexe », ce qui paraît logique. Nous pouvons, bien entendu, en fabriquer une de toute pièce, mais il faudrait alors se poser la même question pour les entiers « long », les entiers « court », etc. ce qui demanderait beaucoup de travail annexe.

Ainsi, nous pouvons proposer une conversion explicite pour changer le réel en un nombre complexe avant d'effectuer l'opération d'affectation. Dans ce cas là, nous devons passer par une création du nombre « Complexe ». En effet, avant, nous avons un réel, et ensuite, il devient un nombre « Complexe ». La seule solution est donc la construction. Nous avons démontré plus haut, que les constructeurs avec un seul paramètre permettent justement les conversions. Dans le cas du nombre « Complexe », il existe effectivement le constructeur avec un réel comme argument.

```
Complexe c ;
c = Complexe(5.3) ;
```

De chaque côté de l'opérateur « = » nous avons bien deux nombres « Complexe ». Le développeur propose une conversion explicite pour passer de la constante littérale réelle « 5.3 » vers un objet anonyme de type « Complexe ». Grâce à l'opérateur d'affectation, le contenu de l'objet anonyme est ensuite transféré vers l'objet « c ».

Nous avons également vu que le compilateur sait faire des appels implicites aux constructeurs si nécessaires. Ainsi, nous pouvons tout simplement écrire :

```
Complexe c ;
c = 5.3 ;
```

L'écriture est équivalente à celle proposée plus haut, sauf qu'il s'agit cette fois-ci d'un appel 'implicite' au constructeur de la classe « Complexe ». Ce constructeur permet donc de fabriquer un objet temporaire anonyme représentant la constante littérale réelle « 5.3 ».

Quel se passe t-il avec l'écriture suivante ?

```
Complexe c ;
c = 5 ;
```

Cette fois-ci, « 5 » est une constante littérale entière. Vu qu'il n'y a pas de proposition de conversion explicite, le compilateur doit se débrouiller tout seul pour convertir l'entier en un nombre « Complexe ». Toutefois, il n'existe pas de constructeur qui permet de réaliser cette opération, la seule possibilité étant le passage d'un réel vers un complexe. Il faut qu'il y ait alors un mécanisme intermédiaire qui éventuellement passe d'un entier vers un réel. Heureusement, ce mécanisme existe intrinsèquement. Finalement, nous avons, dans l'ordre :
Transformation d'un entier en réel
Appel du constructeur pour convertir le réel en un complexe
Affectation pour changer la valeur de « c ».
C (5.0, 0.0) ← Complexe(5.0) ← double (5.0) ← int (5)

Le rôle des conversions implicites associées à l'opérateur « + ».

En définissant un nouvel opérateur « + » pour la classe « Complexe », nous avons vu qu'il existait différentes approches avec, pour chacune, une solution différente. Il s'agit maintenant de déterminer le meilleurs choix possible afin de répondre à toutes les attentes de l'utilisateur. Ce choix devra tenir compte de la possibilité d'effectuer des opérations avec des types différents, notamment, entre des nombres « Complexe » et des réels, puisque nous venons de découvrir qu'il est toujours possible de passer d'un réel vers un « Complexe ».

Nous allons tout d'abords nous intéresser à la dernière solution mise en œuvre, c'est-à-dire à l'opération « + » membre. En effet, cette dernière solution donne l'écriture la plus simple en proposant, par exemple, un seul paramètre.

Par rapport à ce choix, nous allons tester un scénario susceptible de représenter tous les cas de figures prévus par l'opération « + ».

```
//-----------------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
  void decalage(int dx, int dy);
  Complexe operator+(const Complexe &c);
};
                                   Opérateur membre
//-----------------------------------------------------------
inline Complexe Complexe::operator+(const Complexe &c)
{
  return Complexe(reel + c.reel, imaginaire + c.imaginaire);
}
//-----------------------------------------------------------
```

```
int main()
{
  Complexe c1(2.0, 5.0);
  Complexe c2(-5.6, 1.0);
  Complexe c3;

  c3 = c1 + c2;
  c3 = c1 + 3.8;
  c3 = 9.7 + 5.6;
  c3 = 7.5 + c2;

  return 0;
}
```

Cette addition ne pose pas de problème puisque les deux arguments sont des « Complexe ».

c3 = c1.operator+(3.8) ; Le membre de gauche est également un « Complexe ». Il fait donc appel à sa méthode « + » (fonction membre). Le constructeur est sollicité pour proposer la conversion implicite d'un réel vers le nombre « Complexe » équivalent.

Il s'agit ici d'une addition entre deux nombres réels. L'addition proposée par la classe « Complexe » n'est pas du tout concerné par ce problème. En fait, l'addition demandée est celle déjà prévue par défaut par le langage. Par contre, le résultat de cette addition sera évidemment un réel. Du coup, le système mettra en œuvre une conversion implicite afin que l'affectation avec « c3 » se passe correctement.

c3 = (7.5).operator+(c2) ; Cette fois-ci le compilateur n'est pas d'accord. En effet, c'est comme si nous avions un réel qui possède une méthode « + » (fonction membre) qui prend pour argument un nombre « Complexe ». Ce qui n'est évidemment pas le cas par défaut.

Finalement, 'méthode' « + » n'est pas un bon choix. Il est préférable d'utiliser plutôt la 'fonction' « + ». En effet, dans ce cas là, la fonction possède deux paramètres, et l'un ou l'autre peut aussi bien être un « Complexe » qu'un réel. Par ailleurs, l'addition devient parfaitement commutative.

En fait, à chaque fois que vous proposez de surdéfinir un opérateur sur la classe que vous êtes en train de construire, vous devez toujours vous poser cette question : dois-je en faire une méthode ou une fonction ? Dans la mesure du possible, il est préférable d'opter pour la méthode.

```
//-----------------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
  friend Complexe operator+(const Complexe&, const Complexe&);
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
  void decalage(int dx, int dy);
};
//-----------------------------------------------------------
inline Complexe operator+(const Complexe &c1, const Complexe &c2)
{
  return Complexe(c1.reel + c2.reel, c1.imaginaire + c2.imaginaire);
}
//-----------------------------------------------------------
```

L'opérateur "+" est un ami de la classe et, à ce titre, est autorisé à utiliser directement les attributs

Nous pouvons de nouveau atteindre directement les attributs

Les conversions définies par l'utilisateur

Les constructeurs avec un seul paramètre permettent d'assurer des conversions implicites pour aller du type défini par le paramètre vers la classe que nous sommes en train de construire. Ce qui pourrait être intéressant, c'est de proposer l'opération inverse, c'est-à-dire, de réaliser une conversion implicite de la classe concernée vers un type quelconque.

Ce traitement existe. Il est considérer comme une opération au même titre que le symbole « + ». Au lieu de proposer un symbole, il suffit de placer le type de destination à la suite du mot réservé « operator ». A ce sujet, vous avez le droit de prendre n'importe quel type, soit un type primitif, soit un type que vous avez vous-même créé comme, par exemple, une classe.

```
//-----------------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
  friend Complexe operator+(const Complexe&, const Complexe&);
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
  void decalage(int dx, int dy);
  operator double();
};
//-----------------------------------------------------------
Complexe::operator double()
{
  return reel;
}
//-----------------------------------------------------------
```

Conversion implicite pour passer d'un nombre Complexe vers un réel

```
int main()
{
  Complexe c1(2.0, 5.0);
  double r = c1;
}
```

« r » est un réel alors que « c1 » est un « Complexe ». Nous avons donc la conversion implicite qui est sollicité. La partie réelle du nombre complexe est envoyée à « r ».

Une fonction de conversion doit être une fonction membre (méthode). Sa déclaration ne spécifiera aucun type de retour ni de liste de paramètres.

Interdire les conversions implicites – mot réservé : explicit

Nous venons de découvrir qu'un constructeur qui possède un seul paramètre unique a un statut particulier. En effet, il sert, dans ce cas précis, d'opérateur de conversion pour passer du type défini par le paramètre, vers le type de la classe que représente le constructeur.

Dans la majorité des cas, cette situation est très intéressante puisque elle permet d'effectuer les conversions nécessaires attendues. Dans le sujet précédent, nous avons pu, par exemple, passer d'un réel vers un nombre complexe, ce qui paraît normal puisque les réels font partis des nombres complexes.

Il existe des situations, toutefois, où ces conversions implicites sont gênante, et sont quelquefois sources d'erreur difficiles à détecter.

Prenons l'exemple ci-contre que nous avons déjà rencontré lors de l'étude des destructeurs. Nous avons un constructeur avec un seul paramètre unique de type entier non signé.

Il semble ici qu'une conversion implicite d'un entier vers un tableau n'ait pas vraiment de sens. Bien que l'utilisation que l'on en fait généralement ne doive pas poser de problème, il conviendrait tout de même d'interdire cette conversion implicite.

```
class Notes {
  double *notes;          Pointeur de double
  unsigned nombreNotes;   Nouvel attribut
  unsigned taille;        pour gérer la dimension
public:
  Notes(unsigned taille) {
    nombreNotes = 0;
    notes = new double[this->taille = taille];
  }                       Création du tableau dynamique
Destructeur
  ~Notes()  { delete[] notes; }
                          Destruction du tableau dynamique
```

Il suffit alors de préfixer le constructeur du mot réservé explicit. (explicit ne peut être appliqué qu'à un constructeur). La signature du constructeur devient la suivante :

explicit Notes(unsigned taille) ;

Utilisation :

Notes notes(5) ; → oui, construction explicite.
Notes notes = 5 ; → non, construction implicite.
…
notes = 3 ; → non, mise en place de la conversion implicite qui a été interdite.
notes = Notes(3) ; → oui, écriture explicite. Affectation entre deux objets de même type.

Règle sur explicit

Par défaut, les constructeurs permettent de réaliser des conversions implicites lorsque cela est nécessaire. Il peut arriver que dans certaines situations cette conversion automatique soit gênante. Nous pouvons alors bloquer le comportement par défaut du constructeur en demandant que l'argument passer au constructeur au moment de la création de l'objet soit rigoureusement du type attendu. Pour offrir cette alternative, il est nécessaire de préfixer le constructeur du mot réservé « *explicit* ». Lorsque nous avons un constructeur avec un seul paramètre, il est possible d'utiliser l'opérateur « = ». Si vous déclarez un constructeur de type « explicit », cette opportunité n'est plus possible (« = » → création implicite de l'objet). Il est alors nécessaire d'utiliser systématiquement les parenthèses.