Surcharge des opérateurs

1. Généralités

- En langage C, l'opérateur division « `/` » est défini comme suit:
  - `3/2` → Division de deux valeurs entières
  - `3.0/2.0` → Division de deux valeurs réelles
- L'opérateur « `/` » a été redéfini afin de réaliser deux types de division: une entière et une réelle.
- En langage C++, il est possible de redéfinir cet opérateur pour les classes.

Pourquoi?

- Afin d'assurer une meilleure intégration des classes au programme.
- Afin de permettre de fournir aux utilisateurs des interfaces plus simples à manipuler.

```
class Compte{
public:
    Compte Add (const Compte& a, const Compte& b);
    Compte Mul (const Compte& a, const Compte& b) ;
};
 Compte D = Add(Add(Mul(A,B),Mul(B,C)),Mul(C,A)); // beurk!
 Compte D = A*B + B*C + C*A ; // nettement meilleure !
```

Comment?

- Les opérateurs ont tous le même préfixe : « `operator` ».

- Ainsi donc, surcharger l'opérateur « `/` » consiste à définir la méthode « `operator/` ».
- Cette méthode désigne la fonction à 2 opérandes associée à l'opérateur « `/` ».

- Un des opérandes doit être de type classe. Il est donc impossible de redéfinir l'opérateur « `/` » dans une opération « `3/2` » vu que les nombres sont des entiers (type primitif).

- Les arguments ne peuvent pas avoir des valeurs par défaut.

- La priorité et l'associativité des opérateurs ne changent pas (standard).

- Les opérateurs peuvent être définis comme membres ou non membres d'une classe.

Qui peut être redéfini?

- On peut redéfinir une quarantaine d'opérateurs (**1**) :

  - `+ - * / % ^ & |` (1)

`->`	opérateur de sélection d'un membre via pointeur
`[]`	opérateur d'indexation
`()`	opérateur d'appel de fonction
`new`	opérateur d'allocation de mémoire dynamique
`delete`	opérateur de désaffectation de mémoire dynamique

(**1**) La liste complète est disponible sur cette page web:
http://en.wikipedia.org/wiki/Operators_in_C_and_C++

Qui ne peut pas être redéfini?

- Ne peuvent être surchargés les opérateurs suivants (une liste non exhaustive):

`.`	opérateur de sélection d'un membre via objet
`.*`	opérateur pointeur vers un membre via objet
`::`	opérateur de résolution de portée
`?:`	opérateur conditionnel ternaire
`sizeof`	opérateur déterminant la taille en octets

2. Surcharge des opérateurs dans le cadre des classes

- Soit la classe « `Compte` » :

```
class Compte{
    double actif ;
public:
    // etc.
};
```

- Si « `C` » est une instance du type « `Compte` », on désire réaliser les deux opérations suivantes:
  - Ajouter directement à la variable « `actif` » une valeur donnée (ici 123.45),

```
C+= 123.45 ;
```

  - Afficher les données membres de « `Compte` » sur la sortie standard,

```
cout << C ;
```

- Pour ce faire, nous devons surcharger les opérateurs « += » et « << » de la classe compte.

3. Mécanisme de surcharge

- Un opérateur est une fonction (méthode) en C++.
- Une écriture équivalente à « `C+=123.45` » est l'instruction « `operator+=(C,123.45);` »
- Une écriture équivalente à « `cout << C` » est l'instruction « `operator<<(cout,C);` »
- Un opérateur peut être défini:
  - Comme une fonction non membre de la classe à un ou deux arguments:

```
operator+=(C,123.45);
```

  - Comme une fonction membre de la classe avec un argument de moins:

```
C.operator+=(123.45);
```

où `operator+=(double)` est une fonction membre de la classe « `Compte` »

4. Étude du cas de l'opérateur « += »

4.1. Comme fonction non membre de la classe

```
class Compte {
     double actif;
public:
     // etc.
     // surcharge de l'opérateur +=
     friend void operator+=(Compte&,double);
};

// L'opérateur += a été déclaré « friend » afin
// de lui permettre d'accéder aux données « private »
// de la classe « Compte ».

void operator+=(Compte& c, double d){
     c.actif += d;
}

// « X » est une instance de « Compte », les deux
// appels suivants sont équivalents

X+=10.50;
operator+=(X,123.45);
```

4.2. Comme fonction membre de la classe

```
class Compte {
     double actif;
public:
     // etc.
     // surcharge de l'opérateur +=
     void operator+=(double);
};

// L'opérateur += est maintenant une fonction membre
// de la classe « Compte ».

void Compte::operator+=(double d){
     actif += d;
}

// « X » est une instance de « Compte », les deux
// appels suivants sont équivalents

X+=10.50;
X.operator+=(123.45);
```

5. Étude du cas de l'opérateur « << »

- Une classe peut avoir une méthode « `affiche` » très utile pour afficher en sortie les valeurs associées aux membres de la classe.

- Pourquoi ne pas surcharger l'opérateur de sortie « `<<` » afin de remplacer la fonction « `affiche` »?

5.1. Comme fonction non membre de la classe

```
class Compte {
     double actif;
public:
     // etc.
     // surcharge de l'opérateur <<
     friend ostream& operator<<(ostream&,const Compte&);
};

// L'opérateur << a été déclaré « friend » afin
// de lui permettre d'accéder aux données « private »
// de la classe « Compte ».

ostream& operator<<(ostream& out, const Compte& c){
     out << "actif: " << c.actif << endl;
     return out;
}
```

- L'opérateur « `<<` » a été déclaré comme étant ami de la classe « `Compte` » pour lui permettre d'accéder aux données « `private` » de la classe « `Compte` ».

- La méthode « `cout` » est définie dans la classe « `ostream` » (output stream ou flux de sortie), ce qui explique pourquoi « `out` » a été déclarée sous cette forme.

- On passe l'objet « `c` » comme une constante, vu que l'on ne fait qu'afficher le contenu de « `c` », et rien d'autre.

- On retourne un « `ostream` » pour permettre l'enchaînement des opérations sur la sortie standard (en cascades) par exemple:

```
cout << "1er op. vers cout" << "2e op. vers cout" << "etc..";
```

```
Si X est une instance de Compte dont l'actif est 200.89 :

cout << X; // actif: 200.89
```

5.2. Comme fonction membre de la classe

- Si l'on doit écrire en C++ l'opération « `a+=b` », et si l'opérateur est membre de la classe « `a` », cela se traduit par :

```
a.operator+=(b);
```

- Il sera de même, sous les mêmes conditions, pour décrire l'opération « `a<<b` » :

```
a.operator<<(b);
```

- De ce fait, il faut que l'opérateur « `<<` » soit défini dans la classe « `a` » pour que cette opération puisse avoir lieu.

- Prenons l'exemple suivant:

  - Si « `cout` » est une instance de « `ostream` » et « `b` » est une variable du type double,

```
cout << b; revient à écrire cout.operator<<(b);
```

- Si cette opération est autorisée en C++, c'est parce que dans la classe « `ostream` » (le type de « `cout` ») il a été inclus la déclaration suivante:

```
class ostream {
/* etc. */
public:

     ostream& operator<<(double);

/* etc. */
};
```

- Or, nous avons déjà montré que les types manipulés par le flux en sortie « `cout` » et « `cerr` » sont:

```
char short int long float double char*
```

- En effet, dans la classe « `ostream` », l'opérateur « `<<` » a été surchargé afin de supporter les types « `char, short, int, long, float, double, char*` » et rien d'autre.

- Donc rien n'a été prévu dans la classe « `ostream` » pour supporter par exemple le type « `Compte` »!

Que faire alors?

- On ne peut pas intervenir sur la classe « `ostream` » pour inclure le type « `Compte` ». La classe « `ostream` » ne peut pas être modifiée.

- Puisque c'est ainsi, nous allons d'abord voir si l'on peut redéfinir l'opérateur « << » comme membre de la classe « `compte` », puisque, après tout, c'est son contenu que nous voulons afficher sur la sortie standard.

- En se basant sur l'exemple de l'opérateur « += », pour que l'opérateur « << » soit membre de la classe « `Compte` », il faudra écrire alors:

```
class Compte {
     double actif;
public:
     // etc.
     ostream& operator <<(ostream& out);
};
```

- Cette écriture est traduite en C++ comme suit :

```
     Compte.operator<<(out);

Puis,

     Compte << out;
```

- Cette écriture, même si elle est correcte, viole le style de C++, puisque la forme d'écriture voulue est:

```
out << Compte;
```

et non pas

```
Compte << out;
```

Conclusion

- On ne peut pas redéfinir l'opérateur « << » comme membre de la classe (il en est d'ailleurs de même pour l'opérateur « >> »).

- Pour plus de détails voir : « C++ Effective Object-Oriented Software Construction », par K. Dattatri, Prentice Hall, chapitre 7 (page 336 et +).

6. Fonctionnement du compilateur suite à une surcharge de l'opérateur « << »

```
cout << c ; // « c » une instance de Compte
```

- Le compilateur procède comme suit:

1. Il cherche d'abord une fonction membre de la classe `ostream` qui peut accepter le type `Compte`.
2. Il cherche ensuite:

   2.a Une fonction à deux arguments qui accepte l'appel suivant:

```
operator<<(cout,c);
```

   2.b Sinon, une fonction template (voir le cours sur les templates) qui va générer une fonction acceptant l'appel en -2.a-.

3. S'il ne trouve rien, il signale une erreur de compilation.

7. Limitation de la surcharge des opérateurs

- On ne peut surcharger que les opérateurs qui existent déjà. On ne peut pas en inventer!
    - L'opérateur « `??` » n'existe pas en C++, donc impossible de le redéfinir.
- On doit respecter la parité binaire et unaire.
    - « `a+b` » est une parité binaire alors que « `a++` » est une parité unaire.
    - On ne peut pas utiliser l'opérateur « `++` » pour réaliser l'opération :
        - « `a++b` » on doit écrire « `(a++)+b` » ou « `a+(++b)` »
- On doit respecter la priorité et l'associativité
    - « `a+b*c` »
    - La multiplication (`*`) est exécutée avant l'addition (`+`) même lorsque les opérateurs sont redéfinis.
- Il n'y a pas de liens sémantiques implicites entre opérateurs redéfinis (parce que c'est vous qui donnez la sémantique)
    - « `a+=b` » n'est pas forcément la même opération que « `a=a+b` ».
    - « `a==b` » n'implique pas que « `a!=b` » soit faux!

- Il n'y a pas de commutativité automatique

```
double operator+(Compte& c, double d) {}
Compte x(…);
double d=1234.76;

double z = x+d; // ok les arguments sont: Compte,double

// erreur, car operator+ n'a été défini pour supporter cet ordre
// des arguments: double,Compte
z = d+x;
```

8. Opérateur ami ou membre de la classe ?

- Déclarez un opérateur ami lorsque l'opération est symétrique par exemple:
  - `a+b a-b a*b a/b a==b` etc.
- Déclarez un opérateur membre de la classe lorsque l'opération est asymétrique:
  - `a+=b a-=b a*=b a/=b` etc.

9. « RValue » (R-Value) et « LValue » (L-Value)

9.1. Généralités

- Un « lvalue » est tout élément qui peut-être accessible via son adresse (« & »). Un « rvalue » est l'inverse d'un « lvalue ». Un élément temporaire dont l'existence n'est due que le temps d'évaluer une opération donnée.

```
int i = 3 ;

« i » est une « lvalue », « 3 » est un « rvalue ».

Zone mémoire allouée pour la variable « i » est accessible avec
l'opérateur d'adresse « & » ce qui n'est pas le cas pour « 3 ».

int* p = &i ; //OK
int* t = &3 ; // Erreur

int j = i * 5 ;

« j » est une « lvalue », « i*5 » est un « rvalue ».

i * j = 18 ; // Erreur

Le résultat de l'opération « i*j » est un « rvalue » qui ne
possède pas de mémoire. Vu qu'il n'a pas de mémoire assignée, il
n'est pas possible de lui affecter la valeur « 18 ».
```

- Il faut faire attention dans le cas d'une variable constante.

```
const int i = 3 ;

i = 8 ; // Erreur
```

- La variable « i » est considérée dans un tel cas comme une « rvalue ».

- Le fait que la variable soit à gauche de l'expression ne lui permet pas, dans un tel cas, d'être considérée comme une « lvalue ».

9.2. Les références dans le cas de « RValue » et « Lvalue »

- Une référence représente un alias vers un espace mémoire. Nous avons introduit les références dans le cours IFT1166 : « programmation orientée objet en C++ ».

```
int i = 3 ;

int& j = i ;

La variable « j » est une référence à la variable « i ». Les
deux pointent le même espace mémoire.

j = i = 3 ;
```

- Soit l'exemple suivant :

```
void test(int& ref) {
      cout << "ref: " << ref << endl;
}

void test(const int& ref) {
      cout << "const: " << ref << endl;
}

const int z = 50;
int w = 30;

test(z); // Appel de test(const int&)
test(w); // Appel de test(int&)
test(77); // Appel de test(const int&)
```

- Pour le cas du nombre 77, il s'agit d'un « rvalue ». Or, comme nous l'avons expliqué précédemment, un « lvalue » constant se transforme en « rvalue », le compilateur a choisi donc la méthode qui permettait un tel appel.

- Est-ce qu'il est possible d'avoir une fonction spécifique pour traiter les « rvalue »?

- Depuis le « C++11 », il est possible maintenant de référencer une variable « rvalue ».

- On utilise pour cela l'opérateur « && ».

- Cet opérateur permet au compilateur d'obtenir l'adresse de l'expression en cours de traitement.

```
void test(int&& ref) {
     cout << "rref: " << ref << endl;
}

test(77); // Appel cette fois-ci test(int&&)
```

- On peut utiliser la référence à « rvalue » pour étendre sa vie :

```
int zz = 20;
int&& xx = zz + 30; // xx=50 et extension de la durée de vie

xx+=10; // xx=60, vu l'extension

const int& ww = 100;
ww+=10; // Erreur, vu que la variable ww est constante
```

- Il est possible de convertir une variable « lvalue » vers « rvalue » en utilisant pour cela la méthode « move » disponible dans l'espace de nom « std ».

```
test(std::move(w)); // Appel cette fois-ci test(int&&)
```

- Pour l'exemple suivant :

```
int test() {
     int x = 50;
     return x;
}

int& w = test(); // Erreur
int&& z = test(); //OK
```

- La valeur retournée par la fonction « test » est une « rvalue ».

- Une référence à une « rvalue » peut donc capturer la valeur retournée par la fonction. Une référence sur une « lvalue » ne pourra pas le faire.

- Le compilateur « gcc 7.2 »va générer cette erreur :

```
error: cannot bind non-const lvalue reference of type ‘int&’
to an rvalue of type ‘int’
```

- Une référence du type « lvalue » est un alias d'un espace mémoire qui existe, ce qui n'est pas le cas d'un « rvalue ». Ceci explique l'erreur générée par le compilateur.

10. Méthodes par défaut d'une classe

- Sous « C++2003 », chaque classe a par défaut les éléments suivants :

  - Un constructeur par défaut (sans argument)
  - Un constructeur de recopie par défaut
  - Un destructeur par défaut
  - Un opérateur d'affectation par défaut

```
class bidon {
     // membres privés
public:
     bidon(); // un constructeur
     bidon(const bidon&); //un constructeur de recopie
     ~bidon(); // un destructeur
     // un opérateur d'affectation
     bidon &operator=(const bidon&);
};
```

- Il est conseillé de redéfinir le constructeur de recopie et l'opérateur d'affectation si la classe dispose de pointeurs sur des parties dynamiques.

- Ces redéfinitions permettent donc une copie profonde des éléments dynamiques et évitent d'éventuelles fuites de mémoire.

- Le standard « C++11 » a introduit dans une classe quelconque deux nouvelles méthodes par défaut : constructeur par déplacement (move constructor) et l'opérateur d'affectation par déplacement (move assignment operator).

- La précédente classe sera complétée comme suit :

```
class bidon {
     // membres privés
public:
     bidon(); // un constructeur
     bidon(const bidon&); //un constructeur de recopie
     bidon(bidon&&); // un constructeur de déplacement
     ~bidon(); // un destructeur
     // un opérateur d'affectation
     bidon& operator=(const bidon&);
     // un opérateur d'affectation par déplacement
     bidon& operator=(bidon&&);
};
```

- C'est le passage par référence qui permet de distinguer le constructeur de recopie du constructeur de déplacement.

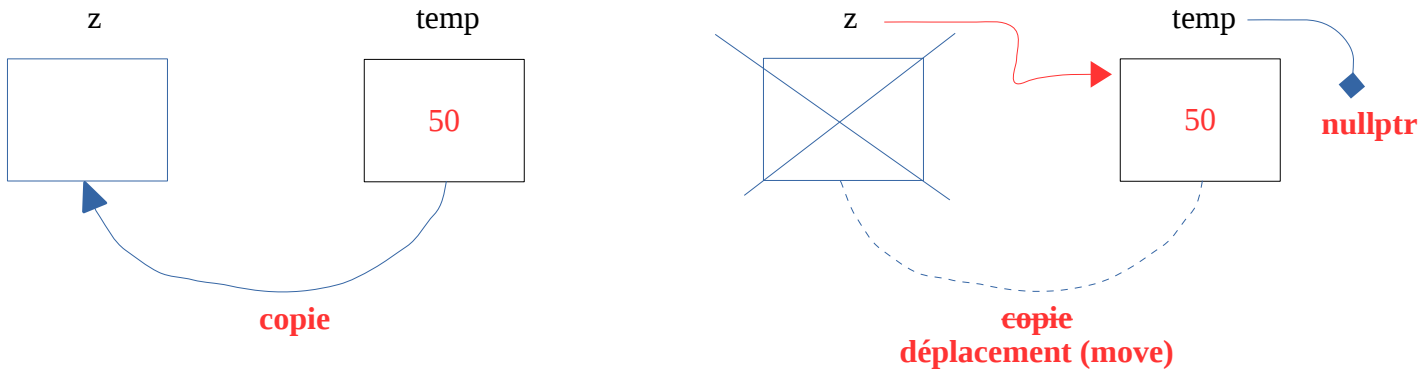
- Le premier utilise une référence sur « lvalue » alors que le second utilise une référence sur « rvalue ».

- Il est de même pour les deux opérateurs d'affectation de la classe.

- L'appel du constructeur de déplacement n'est pas automatique et va dépendre de plusieurs facteurs : implémentation par le compilateur, le niveau d'optimisation surtout au niveau de la valeur de retour (return-value-optimization « RVO »), etc.

- On peut forcer l'appel en utilisant la méthode « move » disponible dans « std », comme nous l'avons utilisée au niveau de la « rvalue ».

- Si on veut copier le contenu de la variable « temp » dans « z » :



- Dans une copie, il faut commencer par créer l'espace associé à « z », pour y copier le contenu de temp.

- Dans un déplacement, il n'est pas nécessaire de créer l'espace mémoire, il faut pointer la variable directement sur l'espace déjà disponible. Il ne faut pas oublier de faire le ménage en réinitialisant les variables non utilisées.

- Nous allons interchanger les valeurs de deux tableaux en utilisant ces nouvelles techniques de déplacement.

- Pour rappel, interchanger deux variables (swap) a et b, consiste à :

```
// swap(a,b) ; a et b des int

swap(int a, int b){
      int temp = a; // temp va préserver le contenu de a
      a = b; // on copie le contenu de b dans a
      b = temp; // on restitue l'ancienne valeur de a dans b
}
```

- Imaginer la même opération entre deux tableaux volumineux! Il faut être très patient! Sans mentionner le cas, ou la taille des deux tableaux n'est pas la même!

```
// une instance de la classe tableau dont les éléments
// sont des entiers
tableau a(50), b(70);
swap(a,b);
```

- Il faudra créer un tableau temporaire ayant la même taille que l'objet « a » pour y copier les éléments de « a ». Détruire l'ancien tableau « a » pour y créer un nouveau tableau de la taille de « b » pour y copier les éléments de « b ». Finalement, détruire l'ancien tableau « b » afin de le remplacer par le tableau temporaire créé au début de cette opération.

- C'est typiquement pour ce genre d'opérations que la notion de déplacement devient intéressante.

- À la fin de l'opération de swap, les données seront les mêmes. Ce qui va changer, c'est leur appartenance. On peut donc juste en manipulant les adresses des tableaux crées éviter de faire des opérations de copie intermédiaires fastidieuses.

- Au final le swap entre deux tableaux sera comme suit :

```
// swap est membre de la classe tableau
void swap(tableau& a, tableau& b){
     tableau temp = std::move(a);
     a = std::move(b);
     b = std::move(temp);
}

tableau a(50), b(70);
swap(a,b);
```

- On commence par déplacer le contenu de « a » dans « temp » et on met à zéro les éléments de « a ». On déplace par la suite les éléments de « b » vers « a » puis on met les éléments de « b » à zéro. Finalement, on fait de même entre « temp » et « b ».

- Ces 3 opérations n'ont pas nécessité d'allocation mémoire pour faire le swap. Elles avaient juste besoin d'ajuster le pointeur d'entrée vers chaque tableau et de s'assurer que les éléments sont remis à zéro avant d'être réutilisés.

- Pour la classe « tableau », le constructeur de déplacement et l'opérateur d'affectation par déplacement sont comme suit :

```
// Constructeur de déplacement (move constructor)
tableau(tableau&& T){
     mData = T.mData;              // copie de l'adresse mémoire
     mTaille = T.mTaille;          // copie de la taille
     T.mData = nullptr;            // ràz de l'ancienne adresse
     T.mTaille = 0;                // ràz de l'ancienne taille
}

// opérateur= de déplacement (move assignment)
tableau& operator=(tableau&& T) {
     if (this != &T){              // on déplace le même elt ?
          delete mData[];          // libère l'ancien espace
          mData =  T.mData;        // copie de l'adresse mémoire
          mTaille = T.mTaille;     // copie de la taille
          T.mData = nullptr;       // ràz de l'ancienne adresse
          T.mTaille = 0;           // ràz de l'ancienne taille
     }
     return *this;                 // on valide les changements
}
```

- Le test « this != &T » n'est pas nécessaire pour un tel opérateur, sauf qu'on le met quand même.

11. Opérateur d'indexation « [] »

11.1. Généralités

- Si « `a` » est un objet alors :
    - L'instruction « `a[expr]` » est équivalente à « `a.operator[](expr)` »
- L'opérateur `[]` ne peut être redéfini qu'avec une fonction membre.

11.2. Exemple d'une classe tableau

```
class tableau {
     // etc.
public:
     // etc.
     double& operator[](int i){
          if ((i>=0) && (i<n)) return tab[i];
          else {
               cerr << "problème avec l'index!\n";
               exit(1);
          }
     }
};
```

```
int main() {

    tableau a(3);
    a.init_tab(10.5);

    // Accès direct aux membres des données private de la classe
    // « tableau » grâce à l'opérateur d'indexation « [] », on peut
    // accéder directement aux éléments du tableau tab de
    // l'objet « a », dans ce cas, on change la valeur de « a[0] ».
    // C'est l'équivalent aussi de « a.operator[](0) »
    // ou bien « a.tab[0] ».

    a[0] = 38.5;

    cout << "on affiche a ...\n";
    cout << a;
    tableau b(2);
    b = a;

    // Même remarque que pour a[0].
    b[1] = 44.44;

    cout << "le tour de b=a ...\n";
    cout << b;

    return 0;
}
```

11.3. Bogue généré par l'opérateur []

- Prenons l'exemple suivant :

```
const tableau s(10); // un tableau de 10 éléments.
```

- Nous avions montré dans le chapitre "Les Propriétés des fonctions membres" (niveau -2-), que les objets constants doivent être manipulés par des fonctions membres constantes. De ce fait :

```
class tableau {
// etc.
public:
    double& operator[](int i) const;
// etc.
};

double& tableau::operator[](int i) const{
    if ((i>=0) && (i<n)) return tab[i];
    else {
         cerr << "problème avec l'index!\n";
         exit(1);
    }
}
```

- On constate que pour une instruction de la sorte :

```
    s[5] = 1.234;
```

- Même si le tableau `s` est constant, il y aura quand même affectation dans `s[5]` de la valeur `1.234`!
- Pour remédier à ce problème,
- nous devons redéfinir **deux** opérateurs d'indexation :

```
class tableau {
// etc.
public:
    double operator[](int i) const; // constant
    double& operator[](int i); // non constant
    // etc.

};
```

```
double& tableau::operator[](int i) {
     if ((i>=0) && (i<n)) return tab[i];
     else {
          cerr << "problème avec l'index!\n";
          exit(1);
     }
}
double tableau::operator[](int i) const{
     if ((i>=0) && (i<n)) return tab[i];
     else {
          cerr << "problème avec l'index!\n";
          exit(1);
     }
}
int main() {
     // un tableau quelconque
     tableau a(3);
     // appel de: double& operator[](int i);
     a[0] = 23.5;
     // un tableau constant
     const tableau s(10);

     // Appel de: double operator[](int i) const;
     // Cet appel retourne que la valeur et ne touche pas au contenu
     // Si dans s[3] il y avait déjà 210.23 => 210.23 = 2345.89
     // Erreur de compilation, affectation impossible !
     s[3] = 2345.89;
     return 0;
}
```

12. Opérateur d'appel de fonction « () »

12.1. Généralités

- Il doit être une fonction membre,
- Il peut prendre autant d'arguments qu'on veut du type manipulé,
- Il peut être redéfini plusieurs fois,
- Il peut prendre des arguments par défaut (ce qui n'est pas le cas des autres opérateurs).

12.2. Exemple d'une classe tableau

```
class tableau {
// etc.
public:
    double operator()(int i,char* f) { …… }
    // redéfini …
    int operator()(double j,int k) { … … … }
    // etc.
};
```

```
int main() {

    tableau t(3);

    double x = t(3,"Fred"); // x=t.operator()(3,"Fred");

    return 0;
}
```

13. Les opérateurs post/pré incrémentation (respectivement décrémentation) « ++ -- »

- Nous allons décrire le cas de l'opérateur post/pré incrémentation « `++` »,
- Les mêmes remarques s'appliquent pour l'opérateur post/pré décrémentation « `--` ».

- Il y a deux formes; préfixe et suffixe.

  - Préfixe → `++a` : incrémente « `a` » et retourne le **nouveau** « `a` ».
  - Suffixe → `a++` : incrémente « `a` » et retourne l'**ancien** « `a` ».

	`operator++` préfixe	`operator++` suffixe
global	`type operator++(type)`	`type operator++(type,int)`
classe	`type opertaor++()`	`type operator++(int)`

- « `a++` » signifie « `operator++(a,0)` » ou bien « `a.operator++(0)` » (idem que pour « `a--` »)

```
#include <iostream>

using namespace std;

class Compte {
     float solde;
public:
     Compte(float m):solde(m) {}
     Compte operator++();
     Compte operator++(int);
    friend ostream& operator<< (ostream& out,const Compte& T);
};

Compte Compte::operator++() {
     solde++;
     return (*this);
}
Compte Compte::operator++(int n) {
     Compte temp = *this;
     solde++;
     return temp;
}

ostream& operator<< (ostream& out,const Compte& T) {
    out << "Solde: " << T.solde << endl;
    return out;
}
```

```
int main() {
     compte C1(100);
     compte C2(200);
     compte C3(300);

     C3 = C1++; // <-- postfixe

     cout << C1; // solde: 101
     cout << C3; // solde: 100

     C3 = ++C2; // <-- préfixe

     cout << C2; // solde: 201
     cout << C3; // solde: 201
     return 0;
}
```