Les opérateurs

Le principe est de pouvoir utiliser les classes pour des opérations ressemblant à des opérations arithmétiques. Par exemple on voudrait pouvoir écrire « prime1 = prime2 » et ainsi donner les valeurs de prime2 à prime1. Le principe reste le même pour les opérateurs tels que + ou -. Si on fait « prime1 – 20 » on doit pouvoir soustraire 20 à la variable membre « MontantBase » de « prime1 ».

Pour redéfinir un opérateur, il suffit d'utiliser le prototype suivant :

```
Type_renvoyé   operator X (type operande) ;
```

Par exemple,

```
PrimeAssurance operator=(PrimeAssurance pr);
```

Redéfinira l'action à accomplir avec l'opérateur "=" lorsque le compilateur rencontrera « une classe = une autre classe » (p3 = p2).

On nous demande d'effectuer les surcharges d'opérateur pour les cas suivants :

Contexte : 3 variables « p » de type classe « PrimeAssurance » sont déclarées et initialisées (utilisation du constructeur d'initialisation :
`p(120.00, 21, 6), p2(250, 14, 10), p3;`)

1) Modification du « montantBase » et affichages

```
p3 = p2; // égalité
p = p + 50.0; // montant de la prime de base augmenté de 50.00
p3 = p2 – 20; // montant de la prime de base diminué
p2 = p + p2; // addition des 2 montants de base, taux TVA le + élevé, bonus malus le + petit

cout << p; cout << p2; // affichage avec opérateur <<
if (p2 < p3) p2 = p3; // comparaison effectuée sur le montant réel de la prime
```

2) modification du degré bonus-malus :

```
cout << ++p << endl;      // préincrément : objet retourné est incrémenté AVANT.

cout << p2++ << endl; // post incrément : objet retourné puis incrémenté APRES.

cout << p2 << endl; // affichage avec <<
```

La surcharge de l'opérateur =

On voudrait effectuer **« p3 = p2 ».**

```
PrimeAssurance operator=(PrimeAssurance pr);
```

Est défini dans « PrimeAssurance.h »

Explication :

- Type « PrimeAssurance » : On doit faire en sorte que p3 soient la copie de p2, donc on renverra une valeur de type « PrimeAssurance »
- Operator= : on redéfinit l'opérateur « = »
- PrimeAssurance pr : le paramètre qui est en fait l'opérande à droite de l'opérateur « = » c'est-à-dire « p2 » .

En ce qui concerne la fonction elle-même, déclarée dans « PrimeAssurance.cxx »

```
PrimeAssurance PrimeAssurance::operator=(PrimeAssurance pr)
{
     cout << "-- SURCHARGE = --" << endl;

     montantBase = pr.montantBase;
     pourcentageTVA = pr.pourcentageTVA;
     nvBonusMalus = pr.nvBonusMalus;

     return *this;
}
```

Le but est d'assigner à la variable actuelle (qui est en fait la variable « p3 ») les valeurs de p2.
On modifie ici de manière classique, comme une copie de valeurs en fait.
Ensuite, on retourne la variable actuelle (p3, dont les variables *« montantBase », « pourcentageTVA » et « nvBonusMalus »* viennent d'être modifiées), pointée par le pointeur **this.*

En résumé :

p3	=	p2
➔ opérande 1		➔ opérande 2 (« pr » dans lfonction)
montantBase	=	pr.montantBase
pourcentageTVA	=	pr.pourcentageTVA
nvBonusMalus	=	pr.nvBonusMalus

```
(2) ***** Test de l'operateur + --> p3 = p2 + 50.0 *****
-- CONSTRUCTEUR DE COPIE --
-- SURCHARGE + --
-- CONSTRUCTEUR DE COPIE --
-- CONSTRUCTEUR DE COPIE --
-- SURCHARGE = --
-- CONSTRUCTEUR DE COPIE --
--> Voici p3 :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 300
Pourcentage de la TVA: 14
Niveau bonus-malus: 10
--> Montrons que p2 n'a pas change :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 250
Pourcentage de la TVA: 14
Niveau bonus-malus: 10
```

La surcharge de l'opérateur +

Le but étant d'effectuer ceci `p3 = p2 + 50.0 // montant de la prime de base augmenté de 50.00`

Le principe reste le même.

50.0 est de type float, et la seconde opérande dans le calcul.

« p » est la première opérande, elle ne sera pas défini comme paramètre

Que faut-il en déduire comme prototype ?

```
PrimeAssurance operator+(float nb)
```

nb est 50.0 dans ce cas-ci, automatiquement passé dans la fonction. Comme pour l'opérateur =, on retourne une variable de type classe PrimeAssurance.

Ce que va faire la fonction:

```
PrimeAssurance PrimeAssurance::operator+(float nb)
{
	PrimeAssurance p(*this);

	cout << "-- SURCHARGE + --" << endl;

	p.montantBase = p.montantBase + nb;
	return p;
}
```

Nouveauté : une nouvelle variable *« p ».* En fait, on va copier la prime d'assurance pointée par `*this` (c'est-à-dire la variable p2 hors de la fonction) dans *p* grâce au constructeur de copie, pour éviter de changer la variable *« p2 »* (variable hors fonction). En fait, la surcharge effectue « `p2 + 50` » seulement.

On fait alors « `p.montantBse=p.montantBase+nb` » puis on retourne « p ». A ce niveau là, une autre surcharge survient :

p3 **=** p2 + 50. La valeur de (p2 + 50) sera copiée dans p3 (voir Surcharge =)

```
(2) ***** Test de l'operateur + --> p3 = p2 + 50.0 *****
-- CONSTRUCTEUR DE COPIE --
-- SURCHARGE + --
-- CONSTRUCTEUR DE COPIE --
-- CONSTRUCTEUR DE COPIE --
-- SURCHARGE = --
-- CONSTRUCTEUR DE COPIE --
--> Voici p3 :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 300
Pourcentage de la TVA: 14
Niveau bonus-malus: 10
--> Montrons que p2 n'a pas change :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 250
Pourcentage de la TVA: 14
Niveau bonus-malus: 10
```

La surcharge de l'opérateur -

Le principe est le même **(p3 = p2 – 20)**

```
PrimeAssurance operator-(float nb);
```

On va vérifier que le montant de la prime (de p2) à diminuer n'est pas inférieure à la valeur qu'on lui soustrait (20).

```
PrimeAssurance PrimeAssurance::operator-(float nb)
{
     int st=0;
     PrimeAssurance p(*this);

     cout << "-- SURCHARGE - --" << endl;

     if (p.montantBase <= nb)
          st = 1;
     if (st==0)
          p.montantBase = p.montantBase - nb;
     return p;
}
```

On utilise ici aussi une variable « p » pour éviter de toucher à « p2 ».
Condition de vérification par rapport au nombre : on place « st » à 1 si jamais le « nombre » est plus grand que le montant de base. Dans ce cas, on passe à la condition suivante qui vérifie si « st == 0 » et si c'est le cas, alors on peut soustraire.
Ensuite, fin de fonction : on retourne « p ».
Après la surcharge de l'opérateur « - » viendra celle de « = ».

```
(4) ***** Test de l'operateur - --> p3 = p2 - 20 *****
-- CONSTRUCTEUR DE COPIE --
-- SURCHARGE - --
-- CONSTRUCTEUR DE COPIE --
-- CONSTRUCTEUR DE COPIE --
-- SURCHARGE = --
-- CONSTRUCTEUR DE COPIE --
--> Voici p3 :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 230
Pourcentage de la TVA: 14
Niveau bonus-malus: 10
--> Montrons que p2 n'a pas change :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 250
Pourcentage de la TVA: 14
Niveau bonus-malus: 10
```

L'exemple est, pour l'opérateur « + » suivant : **p3 = p + p2**
Le but est d'additionner deux classes. On additionne en fait les deux montants de la prime, on choisit le pourcentage de TVA le plus élevé et le niveau bonus malus le plus petit.
« p2 » est l'opérande de droite et sera donc le paramètre passé dans la fonction, dont le prototype est

```
PrimeAssurance operator+(PrimeAssurance pr1);
```

Et la fonction :

```
PrimeAssurance PrimeAssurance::operator+(PrimeAssurance pr1)
{
     PrimeAssurance p(*this);

     p.montantBase = p.montantBase + pr1.montantBase;

     if (p.pourcentageTVA<pr1.pourcentageTVA)
          p.pourcentageTVA = pr1.pourcentageTVA;

     if (p.nvBonusMalus>=pr1.nvBonusMalus)
          p.nvBonusMalus = pr1.nvBonusMalus;

     return p;
}
```

Toujours le pointeur (`*this`) réutilisé et copié dans « p ». (Celui-ci est « p » hors de la fonction, bien entendu)
On additionne le montant de base initial de p avec celui de `pr1.` (Qui est en fait « p2 » hors de la fonction)

Après, des conditions permettant de vérifier quel pourcentage de TVA on choisit et le niveau de bonus malus choisi. Après, on retourne « p » qui sera mis dans « p3 » grâce à la surcharge d'opérateur « = ».

```
(5) ***** Test de l'operateur + --> p3 = p + p2 *****
-- CONSTRUCTEUR DE COPIE --
-- CONSTRUCTEUR DE COPIE --
-- SURCHARGE + --
-- CONSTRUCTEUR DE COPIE --
-- CONSTRUCTEUR DE COPIE --
-- SURCHARGE = --
-- CONSTRUCTEUR DE COPIE --
--> Pour rappel, voici p :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 120
Pourcentage de la TVA: 21
Niveau bonus-malus: 6
--> Pour rappel, voici p2 :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 250
Pourcentage de la TVA: 14
Niveau bonus-malus: 10
--> et voici p3 :
-- FONCTION AFFICHAGE DE LA CLASSE --
Montant de base: 370
Pourcentage de la TVA: 21
Niveau bonus-malus: 6
```

Surcharge <<

On utilise une fonction “amie” si les paramètres sont de type différent ➔ opérateur est dit libre. Dans le cas de « << », on va en fait faire en sorte que « cout << p2 » affiche la classe !

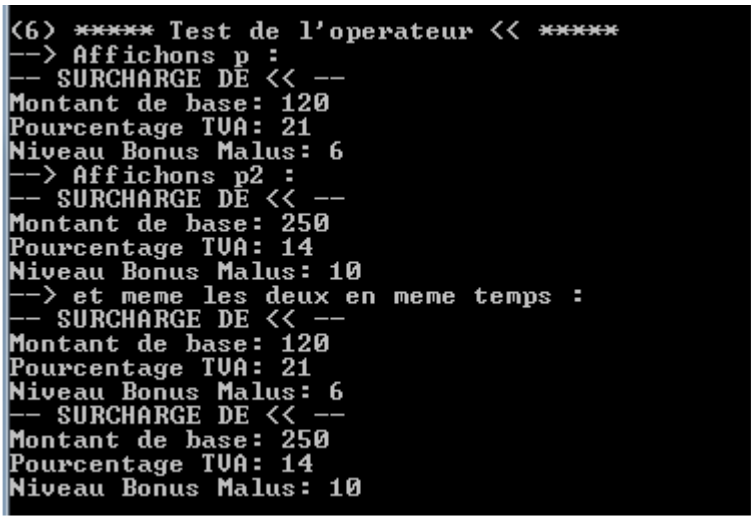
Prototype de la fonction :

```
friend ostream& operator<<(ostream& str, const PrimeAssurance&
```

Et la fonction :

```
ostream& operator<<(ostream& str, const PrimeAssurance& pr)
{
     cout << "-- SURCHARGE DE << --" << endl;
     str << "Montant de base: " << pr.montantBase << "\n" <<
"Pourcentage TVA: " << pr.pourcentageTVA << "\n" << "Niveau
Bonus Malus: " << pr.nvBonusMalus << endl;;
     return str;
}
```

Ici, le « `friend` » n’est pas réutilisé, le compilateur sait s’y retrouver. « `ostream` » est en fait un type « `buffer` ». Le premier paramètre est tout ce qu’il faudra mettre dans le buffer pour afficher. Le deuxième est la classe à afficher dont on récupère les valeurs et qu’on place dans le buffer « `str` » avec des opérateurs d’insertion. On retourne le buffer qui sera affiché par « `cout` » hors fonction.



Surcharge < ou >

On utilise une fonction “amie” si les paramètres sont de type différent ➔ opérateur est dit libre. Dans le cas de « << », on va en fait faire en sorte que « cout << p2 » affiche la classe !

```
  cout << "--> Prime p2 = " << p2.getMontantPrime() << endl;
  cout << "--> Prime p3 = " << p3.getMontantPrime() << endl;
  if (p2 < p3) cout << "La prime p2 est plus petite que p3" <<
endl;
  if (p2 > p3) cout << "La prime p2 est plus grande que p3" <<
endl;
  if (p2 == p3) cout << "La prime p2 est egale a p3" << endl;
  cout << endl;
```

Ici, on affiche le montant calculé de la prime (rappel : fonction « inline » dans le « main ») puis on va comparé la prime 2 et la prime 3.
Les trois opérateurs de comparaison sont « < », « > », et « == ».
Les trois prototypes sont :

```
int operator<(PrimeAssurance pr);
int operator>(PrimeAssurance pr);
int operator==(PrimeAssurance pr);
```

PrimeAssurance pr : l’opérande de DROITE ! (p2 < **p3**)
Les fonctions :

```
int PrimeAssurance::operator<(PrimeAssurance pr)
{
     PrimeAssurance p1(*this);

     cout << "-- SURCHARGE: " << p1.getMontantPrime() << " ? <
que " <<pr.getMontantPrime() << "--" << endl;

     if(p1.getMontantPrime()<pr.getMontantPrime())
          return 1;
     return 0;
}
```

NB : Si la condition est vérifiée, on retourne 1, sinon 0.
`p1` équivaut à « p2 » hors de la fonction
`pr` équivaut à « p3 » hors de la fonction

```
int PrimeAssurance::operator>(PrimeAssurance pr)
{
     PrimeAssurance p1(*this);

     cout << "-- SURCHARGE: " << p1.getMontantPrime() << " ? >
que " <<pr.getMontantPrime() << "--" << endl;

     if(p1.getMontantPrime()>pr.getMontantPrime())
          return 1;
     return 0;
}
```

NB : Si la condition est vérifiée, on retourne 1, sinon 0.
p1 équivaut à « p2 » hors de la fonction
pr équivaut à « p3 » hors de la fonction

```
int PrimeAssurance::operator==(PrimeAssurance pr)
{
     PrimeAssurance p1(*this);

     cout << "-- SURCHARGE: " << p1.getMontantPrime() << " ? ==
a " <<pr.getMontantPrime() << "--" << endl;

     if(p1.getMontantPrime()==pr.getMontantPrime())
          return 1;
     return 0;
}
```

NB : Si la condition est vérifiée, on retourne 1, sinon 0.
p1 équivaut à « p2 » hors de la fonction
pr équivaut à « p3 » hors de la fonction

```
(7) ***** Test des operateurs de comparaison < > == *****
-- OBTENIR LE MONTANT DE LA PRIME --
--> Prime p2 = 522.5
-- OBTENIR LE MONTANT DE LA PRIME --
--> Prime p3 = 671.55
-- CONSTRUCTEUR DE COPIE --
-- CONSTRUCTEUR DE COPIE --
-- OBTENIR LE MONTANT DE LA PRIME --
-- OBTENIR LE MONTANT DE LA PRIME --
-- SURCHARGE: 522.5 ? < que 671.55--
-- OBTENIR LE MONTANT DE LA PRIME --
-- OBTENIR LE MONTANT DE LA PRIME --
La prime p2 est plus petite que p3
-- CONSTRUCTEUR DE COPIE --
-- CONSTRUCTEUR DE COPIE --
-- OBTENIR LE MONTANT DE LA PRIME --
-- OBTENIR LE MONTANT DE LA PRIME --
-- SURCHARGE: 522.5 ? > que 671.55--
-- OBTENIR LE MONTANT DE LA PRIME --
-- OBTENIR LE MONTANT DE LA PRIME --
-- CONSTRUCTEUR DE COPIE --
-- CONSTRUCTEUR DE COPIE --
-- OBTENIR LE MONTANT DE LA PRIME --
-- OBTENIR LE MONTANT DE LA PRIME --
-- SURCHARGE: 522.5 ? == a 671.55--
-- OBTENIR LE MONTANT DE LA PRIME --
-- OBTENIR LE MONTANT DE LA PRIME --
```

Surcharge de ++

Prototypes:

```
PrimeAssurance operator++();
PrimeAssurance operator++(int);
```

➔ Préincrément : retourne l'objet incrémenté ! **(++p)**

```
PrimeAssurance PrimeAssurance::operator++()
{
     (*this).nvBonusMalus = (*this).nvBonusMalus + 1;
     return *this;
}
```

➔ Postincrément : retourne l'objet non incrémenté, puis incrémente ! **(p++)**

```
PrimeAssurance PrimeAssurance::operator++(int)
{
     PrimeAssurance temporaire(*this);
     (*this).nvBonusMalus = (*this).nvBonusMalus + 1;
     return temporaire;
}
```

Toujours la copie de (*this) dans « temporaire » dans ce cas.