

# La surcharge d'opérateurs sous Delphi 2006 Win32

par Laurent Dardenne ([Contributions](#)) >

Date de publication : 12/12/2007

Dernière mise à jour : 12/10/2007

A partir de Delphi 2006 la surcharge d'un opérateur au sein d'un enregistrement est désormais possible. Revenons dans le détail de cette nouvelle possibilité du langage.

Je tiens à remercier **Sébastien Doeraene** pour ses apports et remarques constructives ainsi que pour ces corrections orthographiques.

- 1 - Public concerné
  - 1-1 - Les sources
- 2 - Qu'entend-on par surcharge d'opérateur ?
- 3 - Rappels sur les opérateurs du langage Delphi
- 4 - Remarques concernant les enregistrements avancés
- 5 - Les opérateurs surchargeables
  - 5-1 - Opérateurs unaire
  - 5-2 - Opérateurs binaire
  - 5-3 - Opérateurs de comparaison
  - 5-4 - Opérateurs de conversion
- 6 - La directive inline
- 7 - Règles de priorité des opérateurs
  - 7-1 - Commutativité
- 8 - Exemples proposés

## 1 - Public concerné



Testé sous Xp et Delphi 2006 update 2 - hotfix 10.

### 1-1 - Les sources

Les fichiers sources des différents exemples :

**FTP.**

**HTTP.**

## 2 - Qu'entend-on par surcharge d'opérateur ?

La surcharge d'un opérateur, ou *polymorphisme ad-hoc*, permet de redéfinir son action afin qu'il exécute une fonction spécifiée lorsqu'il est utilisé avec un type d'enregistrement particulier, et ce avec Delphi 2006 sous Win32, Delphi .NET proposant en plus la surcharge d'opérateur pour les classes.


Ainsi l'ajout de 2 enregistrements devient sémantiquement possible :

```
TMonRecord = record
  Champ1: Integer;
  ...
end;

var RecordA, RecordB : TMonRecord;
    Resultat: TMonRecord;
begin
  RecordA.Champ1:=10;
  RecordB.Champ1:=11;
  RecordA:=RecordA+RecordB;
  Resultat:=RecordA+RecordB;
  ...
end;
```

Les opérateurs surchargés remplacent avantageusement les appels de fonction (c'est ce que l'on pourrait appeler du "FAQ sucre syntaxique").

Ce *sucre syntaxique* facilite l'écriture de code pour les utilisateurs de vos enregistrements en simplifiant leur usage.

 *La surcharge de certains opérateurs n'est sensée que pour certains types de données, autant l'addition de points est sensée, on obtient bien un troisième point, autant l'addition de voitures ne l'est pas car ici on n'obtient pas une troisième voiture, à la rigueur un amas de pièces détachés :-). Evitez donc d'additionner des choux et des carottes.*

### 3 - Rappels sur les opérateurs du langage Delphi

Avant de débiter rappelons ce qu'est un opérateur, voici un extrait de l'aide de Delphi à ce sujet :

*"Les opérateurs agissent comme des fonctions prédéfinies faisant partie du langage Delphi. Ainsi, l'expression (X + Y) est construite à partir des variables X et Y (appelées des opérands) et avec l'opérateur + ; quand X et Y représentent des entiers ou des réels, (X + Y) renvoie leur somme.*

*Les opérateurs sont @, **not**, ^, \*, /, **div**, **mod**, **and**, **shl**, **shr**, **as**, +, -, **or**, **xor**, =, >, <, <>, <=, >=, **in** et **is**. Les opérateurs @, **not** et ^ sont des opérateurs unaires (n'utilisant qu'un seul opérande). Tous les autres opérateurs sont binaires (ils utilisent deux opérands), à l'exception de + et - qui peuvent être unaires ou binaires. Un opérateur unaire précède toujours son opérande (par exemple -B) à l'exception de ^ qui suit son opérande (par exemple, P^). Un opérateur binaire est placé entre ses opérands (par exemple, A = 7).*

*Certains opérateurs se comportent différemment selon le type des données transmises. Ainsi, **not** effectue une négation bit-à-bit pour un opérande entier et une négation logique pour un opérande **booléen**. De tels opérateurs apparaissent dans plusieurs des catégories indiquées plus bas. A l'exception de ^, **is** et **in**, tous les opérateurs acceptent des opérands de type **Variant**. Les sections suivantes supposent une certaine connaissance des types de données Delphi. Pour davantage d'informations sur la priorité des opérateurs dans les expressions complexes, voir Règles de priorité des opérateurs..."*

## 4 - Remarques concernant les enregistrements avancés

### Sous Delphi Win32 :

Seuls les enregistrements avancés peuvent utiliser la surcharge d'opérateurs. Voici ce que nous dit l'aide de Delphi à ce sujet :

- *En plus des types d'enregistrements traditionnels, le langage Delphi autorise des types d'enregistrements plus complexes, similaires à des classes.*
- *"En plus des champs, les enregistrements peuvent avoir des propriétés et des méthodes (incluant les constructeurs), des propriétés de classe, des méthodes de classe, des champs de classe et des types imbriqués.*
- *Les enregistrements ne prennent pas en charge l'héritage.*
- *Les enregistrements peuvent contenir une partie variable alors que les classes ne le peuvent pas.*
- *Les enregistrements sont des types valeur, copiés par affectation, transmis par valeur, et alloués sur la pile à moins qu'ils ne soient déclarés globalement ou alloués explicitement au moyen des fonctions New et Dispose. Les classes sont des types référence ; elles ne sont pas copiées par affectations, elles sont transmises par référence et sont allouées sur le tas.*
- *Les enregistrements permettent la surcharge d'opérateurs sur les plates-formes Win32 et .NET ; les classes permettent la surcharge d'opérateurs uniquement pour .NET.*
- *Les enregistrements sont construits automatiquement, en utilisant un constructeur par défaut sans argument, alors que les classes doivent être construites explicitement. Comme les enregistrements ont un constructeur par défaut sans argument, aucun constructeur d'enregistrement défini par l'utilisateur ne doit avoir de paramètre.*
- *Les types enregistrement ne peuvent pas avoir de destructeurs.*
- *Les méthodes virtuelles (celles spécifiées par les mots-clés virtual, dynamic et message) ne peuvent pas être utilisées dans les types enregistrement.*
- *A la différence des classes, les types enregistrement sur la plate-forme Win32 ne peuvent pas implémenter d'interfaces ; toutefois, les enregistrements sur la plate-forme .NET peuvent implémenter des interfaces."*

A quoi j'ajouterais ces remarques :

- L'usage de méthodes de classe dans les records avancés nécessite la directive **static** (dans ce cas le compilateur ne gère pas de paramètre **self** dans les appels de méthode).
- Vous pouvez utiliser au sein d'une déclaration d'enregistrement le nom du type en cours de définition mais uniquement dans les signatures de méthodes. Cela permet de le référencer avant qu'il ne soit effectivement défini.
- La directive **overload** est implicite.
- **Plus d'informations...**

### Sous Delphi .NET 1.1 :

Les classes supportent la surcharge d'opérateurs et les interfaces peuvent être déclarées dans la définition d'un enregistrement. Très peu de classes du .NET Framework ont des opérateurs surchargés, mais la plupart des types valeur en ont.

## 5 - Les opérateurs surchargeables


L'aide en ligne nous propose un tableau contenant les informations suivantes :


Opérateur	Catégorie	Signature de déclaration	Mappage de symbole
Nom de l'opérateur	Catégorie de l'opérateur	Déclaration à respecter pour la méthode d'implémentation	Correspondance du symbole pour cet opérateur

Dans les signatures présentées plus avant les termes *type* et *resultType* renverront le plus souvent au nom du type de votre enregistrement. Notez que les opérateurs de conversion utiliseront forcément d'autres types mais au moins celui du record manipulé.

Ce qui permettra les écritures suivantes, impliquant un type **record** et un type **Integer** :

```
RecordB:=10;
RecordA:=RecordB+10;
```


 -Le compilateur n'impose pas la déclaration combinée des opérateurs. Par exemple la redéfinition de l'opérateur > ne nécessite pas de déclarer l'opérateur associé <. Mais il est préférable de surcharger les opérateurs de manière symétrique.

 -Rien ne vous empêche d'opérer sur plusieurs champs de votre enregistrement, vous seul donnez la cohérence à l'opération effectuée.

Le compilateur quant à lui se charge d'appeler votre opérateur tout en respectant les règles de priorités.

Notez qu'il est tout à fait possible de surcharger plusieurs fois un opérateur :

```
RecordA:=RecordB+10; //Opérateur Add pour un entier
RecordA:=RecordB+'10'; //Opérateur Add pour une chaîne de caractère
RecordA:=RecordB+RecordB; //Opérateur Add pour le type TMonRecord
```

 Les opérateurs surchargés ne peuvent pas être référencés par leur nom de méthode dans le code source.

D'après la documentation de Delphi :

"En règle générale, **les opérateurs ne doivent pas modifier leurs opérands**. A la place, ils renvoient une nouvelle valeur, construite en effectuant l'opération sur les paramètres."

### 5-1 - Opérateurs unaire

Négation	Negative(a: type): resultType;	-
Positive		+

	Positive(a: type): resultType;	
Incrémenter	Inc(a: type): resultType;	<b>Inc</b>
Décémenter	Dec(a: type): resultType	<b>Dec</b>
Négation logique	LogicalNot(a: type): resultType;	<i>Not</i>
Négation bit-à-bit ( <b>bug</b> open)	BitwiseNot(a: type): resultType;	<i>aucun</i>
Tronquer	Trunc(a: type): resultType;	<b>Trunc</b>
Arrondir	Round(a: type): resultType;	<b>Round</b>

Un opérateur unaire précède toujours son opérande (par exemple -B).

Ce sont des opérateurs n'utilisant qu'un seul opérande. Pour ceux-ci, on reçoit un opérande et on renvoie une valeur du type de votre enregistrement puisque l'opération porte sur la même donnée.

```
TMonRecord = record
  Champ1: Byte;
  class operator Negative(const Value: TMonRecord): TMonRecord;
end;
...
class operator TMonRecord.Negative(const Value: TMonRecord): TMonRecord;
begin
  Result.Champ1 := -Value.Champ1;
end;
```

## 5-2 - Opérateurs binaire

Ajouter	Add(a: type; b: type): resultType;	<b>+</b>
Soustraire	Subtract(a: type; b: type): resultType;	<b>-</b>
Multiplier	Multiply(a: type; b: type): resultType;	<b>*</b>
Diviser	Divide(a: type; b: type): resultType;	<b>/</b>
Division entière	IntDivide(a: type; b: type): resultType;	<b>div</b>
Modulo	Modulus(a: type; b: type): resultType;	<b>mod</b>
Décalage à gauche bit-à-bit	ShiftLeft(a: type; b: type): resultType;	<b>shl</b>
Décalage à droite bit-à-bit	ShiftRight(a: type; b: type): resultType;	<b>shr</b>
And logique	LogicalAnd(a: type; b: type): resultType;	<b>and</b>
Or logique		<b>ou</b>

	LogicalOr(a: type; b: type): resultType;	
Xor logique	LogicalXor(a: type; b: type): resultType;	<b>xor</b>
And bit-à-bit	BitwiseAnd(a: type; b: type): resultType;	<b>and</b>
Or bit-à-bit	BitwiseOr(a: type; b: type): resultType;	<b>ou</b>
Xor bit-à-bit (ou exclusif)	BitwiseXor(a: type; b: type): resultType;	<b>xor</b>


Un opérateur binaire est placé entre ses opérandes (par exemple, B+C).

Ce sont des opérateurs qui utilisent deux opérandes. Pour ceux-ci, on reçoit donc 2 opérandes et on renvoie une valeur du type de votre enregistrement puisque l'opération porte sur la même donnée.

```

TMonRecord = record
  Champ1: Byte;
  class operator Add(a: TMonRecord; b: TMonRecord): TMonRecord;
end;


```

 **Attention**, vous devez contrôler les éventuelles impossibilités et dépassement de capacité pour ces types d'opérations.

### 5-3 - Opérateurs de comparaison

Egalité	Equal(a: type; b: type): Boolean;	=
Négation	NotEqual(a: type; b: type): Boolean;	<>
Plus grand que	GreaterThan(a: type; b: type) Boolean;	>
Plus grand ou égale à	GreaterThanOrEqual(a: type; b: type): Boolean;	>=
Inférieur à	LessThan(a: type; b: type): Boolean;	<
Inférieur ou égale à	LessThanOrEqual(a: type; b: type): Boolean;	<=

Un opérateur de comparaison est placé entre ses opérandes (par exemple, B>=C).

 **La documentation de Delphi 2006, concernant les opérateurs de comparaison, semble erronée car elle indique que les trois derniers renvoient resultType.**

L'égalité doit se faire sur le contenu de tous les champs de l'enregistrement.

```

class operator TMonRecord.Equal(a, b: TMonRecord): Boolean;
begin

```

```
Result:=A.Champ1=b.Champ1;  
end;
```

## 5-4 - Opérateurs de conversion

Conversion de type implicite	Implicit(a : type): resultType;	transtypage implicite
Conversion de type explicite	Explicit(a: type): resultType;	transtypage explicite

Ils permettent de transtyper l'enregistrement dans des affectations ou des appels de procédure ou de fonction.

Par exemple le code suivant ne peut compiler sans la surcharge de l'opérateur **Explicit** :

```
Writeln('Resultat=',RecordA);
```

Ce code renvoi lors de la compilation l'erreur suivante :

```
Type illégal dans l'instruction Write/Writeln (E2054)
```

On doit donc déclarer l'opérateur **Explicit** :

```
// Conversion explicite de TMonRecord en Integer  
class operator TMonRecord.Explicit(a: TMonRecord): Integer;  
begin  
  Result:=a.Champ1;  
end;
```

ce qui permet de modifier l'appel ainsi :

```
Writeln('Resultat=',Integer(RecordA));
```

L'exemple suivant ne peut compiler sans la surcharge de l'opérateur **Implicit** :

```
RecordA:=10;
```

Ce code renvoi lors de la compilation l'erreur suivante :

```
Types incompatibles : 'TMonRecord' et 'Integer'
```

Pour une affectation on déclarera l'opérateur de transtypage **Implicit** :

```
class operator TMonRecord.Implicit(a: Integer): TMonRecord;  
begin  
  Result.Champ1:=a;  
end;
```

```
//Ici il s'agit de la conversion implicite du type TMonRecord vers une variable de type Integer
class operator TMonRecord.Implicit(a: TMonRecord): Integer;
begin
    Result:=a.Champ1;
end;
```


Pour rappel ces opérateurs peuvent être définis de nombreuses fois, c'est à dire pour ceux-ci tant que votre enregistrement nécessitera une conversion vers un type particulier.

```
class operator TMonRecord.Implicit(a: TMonRecord): String;
begin
    Result:=IntToStr(a.Champ1);
end;
...
Chaine:="Valeur de l'enregistrement"+RecordA;
```

 Voici ce que conseille Sébastien Doeraene :

*"Implémenter **Implicit** c'est implémenter **Implicit+Explicit**. Mais implémenter **Explicit** ce n'est implémenter que **Explicit**. Il ne faut donc jamais implémenter les deux pour le même type et dans le même sens (valeur -> record VS record -> valeur).*

*Ma philosophie est d'implémenter **Implicit** lorsqu'il n'y a pas de perte de valeur. Par exemple affecter un réel à un complexe est implicite. Et d'utiliser **Explicit** lorsqu'il y a une perte de valeur : l'affectation d'un complexe à un réel serait explicite, et "droperait" la partie imaginaire. Comme le fait l'affectation d'un réel à un entier, qui "droppe" la partie décimale.*

 Si les 2 opérateurs de conversion existent pour un enregistrement donné et renvoi le même type, l'opérateur implicite sera appelé par défaut. Par exemple :

```
TMonRecord = record
    Champ1: Integer;
    //A éviter : Déclarer les 2 opérateurs à l'identique
    // Conversion explicite de TMonRecord en Integer
    class operator Explicit(a: TMonRecord): Integer;
    // Conversion implicite de TMonRecord en Integer
    class operator Implicit(a: TMonRecord): Integer;
end;
```


 **Aucun autre opérateur que ceux présentés ici ne peut être surchargé ni créé.**


## 6 - La directive inline

La définition d'opérateurs se prête bien à l'usage de la directive **inline**. Voyons ce que nous dit l'aide en ligne de Delphi 2006 à son sujet :

*"Pour améliorer les performances, le compilateur Delphi permet le balisage des fonctions et procédures à l'aide de la directive **inline**. Si une fonction ou procédure répond à certains critères, le compilateur insère directement le code au lieu de générer un appel. **L'utilisation de cette directive induit une optimisation des performances pouvant se traduire par un code plus rapide, mais elle présente des inconvénients en termes d'espace.** En effet, elle entraîne toujours la création d'un fichier binaire plus volumineux par le compilateur."*

```
class operator TMonRecord.Negative(const Value: TMonRecord): TMonRecord;inline;
begin
    Result.Champ1:=-Value.Champ1;
end;
```

 Elle peut être gérée via la directive de compilation **{\$INLINE ON/AUTO/OFF}**. De nombreuses fonctions de la VCL utilisent désormais cette directive.

 "L'utilisation de la directive **inline** par le compilateur n'est qu'une suggestion. En effet, il n'existe aucune garantie quant à son utilisation par le compilateur pour une routine particulière car, dans certains cas, cette directive ne peut pas être employée."

Consultez dans l'aide en ligne la liste des situations dans lesquelles la directive **inline** peut ou ne peut pas être utilisée.

Exemple avec **{\$INLINE OFF}** :

```
Project1.dpr.102: Resultat:=RecordA+RecordB;
00409160 8B1524E24000    mov edx,[$0040e224]
00409166 A120E24000    mov eax,[$0040e220]
0040916B E890F8FFFF    call TMonRecord.&op_Addition
00409170 A328E24000    mov [$0040e228],eax
```

Exemple avec **{\$INLINE ON}** :

```
Project1.dpr.102: Resultat:=RecordA+RecordB;
00409160 A120E24000    mov eax,[$0040e220]
00409165 030524E24000    add eax,[$0040e224]
0040916B A32CE24000    mov [$0040e22c],eax
00409170 A12CE24000    mov eax,[$0040e22c]
00409175 A328E24000    mov [$0040e228],eax
```

## 7 - Règles de priorité des opérateurs

Extrait de la documentation de Delphi 2006 :

*Dans des expressions complexes, les règles de priorité déterminent l'ordre dans lequel les opérations sont effectuées.*

### **Priorité des opérateurs**

Opérateurs	Priorité
@, not	première (maximale)
*, /, div, mod, and, shl, shr, as	deuxième
+, -, or, xor	troisième
=, <>, <, >, <=, >=, in, is	quatrième (minimale)

*Un opérateur de priorité plus élevée est évalué avant un opérateur de priorité plus basse, les opérateurs de même priorité étant évalués à partir de la gauche. Ainsi, l'expression :*

$X + Y * Z$

*multiplie Y par Z, puis ajoute X au résultat ; \* est évaluée en premier car sa priorité est supérieure à celle de +. Mais  $X - Y + Z$*

*commence par soustraire Y à X puis ajoute Z au résultat : - et + ayant la même priorité, l'opération de gauche est effectuée en premier. Vous pouvez utiliser des parenthèses pour redéfinir ces règles de priorité. Une expression entre parenthèses est tout d'abord évaluée puis traitée comme un seul opérande. Par exemple,*

$(X + Y) * Z$

*multiplie Z par la somme de X et Y. Les parenthèses sont parfois nécessaires dans des situations où, au premier regard elles ne semblent pas utiles. Par exemple, soit l'expression :*

$X = Y \text{ or } X = Z$

*L'interprétation voulue est manifestement :*

$(X = Y) \text{ or } (X = Z)$

*Néanmoins, sans parenthèses, le compilateur respecte les règles de priorité des opérateurs et l'interprète comme :*

$(X = (Y \text{ or } X)) = Z$

*ce qui provoque une erreur de compilation sauf si Z est un booléen. Les parenthèses rendent souvent le code plus simple à écrire et à lire même quand elles sont, techniquement parlant, inutiles. Ainsi le premier exemple peut également s'écrire :*

$X + (Y * Z)$

Ici, les parenthèses ne sont pas nécessaires pour le compilateur, mais elles épargnent au programmeur et au lecteur la nécessité de réfléchir à la priorité des opérateurs.

## 7-1 - Commutativité

 D'après la documentation de Delphi 2006 :

*"Il n'existe aucune hypothèse concernant les propriétés distributives ou commutatives de l'opération. Pour les opérateurs binaires, le premier paramètre est toujours l'opérande gauche et le second paramètre l'opérande droit. En l'absence de parenthèses explicites, l'associativité est supposée être de gauche à droite."*

Par exemple les instructions suivantes :

```
RecordA:=RecordB+10; //Opérateur Add pour le type integer  
RecordA:=10+RecordB; //Opérateur Add pour le type integer
```

laisse supposer qu'il faille doubler l'implémentation de l'opérateur concerné en inversant la liste de ces paramètres :

```
TMonRecord = record  
  Champ1: Integer;  
  //RecordA:=RecordB+10;  
  class operator Add(a: TMonRecord; b: Integer): TMonRecord; inline;  
  //RecordA:=10+RecordB;  
  class operator Add(a: Integer; b: TMonRecord): TMonRecord; inline;  
  ...
```

Dans notre cas la déclaration de l'opérateur **Implicit** éviterai la seconde déclaration de l'opérateur **Add**. Le compilateur effectuant dans ce cas une conversion implicite **integer** vers **TMonRecord**, via l'opérateur **implicit**, puis appellera l'opérateur **Add**.

Dans certain contexte il reste possible de doubler l'implémentation d'un opérateur si par exemple l'opérateur **Implicit** est dénué de sens.

Par exemple, si on voulait faire un pointeur qui supporte l'arithmétique des pointeurs du C. On implémenterait un **Add(Ptr: TPointer; Value: Integer)** et pas l'Implicit.

## 8 - Exemples proposés

Les records avancés permettent à l'aide des opérateurs de faciliter la manipulation de nouveaux types et/ou de structures de données évoluées.

Le projet **SurchargeOperateurs.dpr** reprend les quelques ligne de code présentées dans ce tutoriel.

L'unité **BigSetExample** propose la gestion d'ensemble autour d'un type de base possédant plus de 256 valeurs.

### Type

```
TCouleur = (coBlanc, coBleu, coRouge, coVert, coNoir);  
TCouleurs = set of TCouleur;
```



*Vous devez adapter les blocs de code assembleur selon que le type utilisé est stocké sur 2 ou 4 octets :*

```
TBigValue = $0000..$FFFF; //Valeur codée sur 2 octets, pas de MOVZX  
TBigValue = $0000..$10000; //Valeur codée sur 4 octets, MOVZX obligatoire
```

A noter l'unité **BigSetD7Older** pour Delphi versions 5 à 7.

L'exemple **EssaisNullable.dpr** propose une solution d'implémentation d'un type integer nullable. La gestion de la valeur *NULL* est simulée à l'aide de **Nil** et en respectant les règles suivantes :

- **nil** n'est jamais égal à **nil** ni différent de **nil**.
- pour toute opération dont au moins un opérande vaut **nil**, le résultat est **nil**.

L'exemple contenu dans le fichier **23581\_win32\_operator\_overloading\_complex\_numbers.ZIP**, provenant de CodeCentral, propose l'implémentation d'un nombre complexe.

