

&

NewGInt & NewGCent

Unités .pas pour la manipulation des grands nombres entiers avec
DELPHI

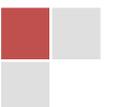


Table des Matières

Genèse	2
Fonctions et procédures de NewGInt	4
Les convertisseurs	4
Les générateurs	5
Les opérations élémentaires	5
Les opérations utilitaires rapides	7
Les décalages	7
Les opérations logiques	8
Les tests	8
Fonctions et procédures de NewGCent	9
Les convertisseurs	9
Les générateurs	10
Les opérations élémentaires	10
Les opérations utilitaires rapides	11
Les décalages	11
Les tests	11
Astuces et conseils communs	12
Conclusion	13
Perspectives	13

N.B.1 Les unités concernées sont mises gracieusement à la disposition de tous à titre expérimental et ni leur utilisation incorrecte ou correcte, ni un dysfonctionnement éventuel des routines qu'elles contiennent ne sauraient entraîner la responsabilité de leurs auteurs en cas de dysfonctionnement avéré des programmes qui les intègrent.

N.B.2 Pour les programmeurs ayant déjà utilisé ou fait connaissance avec l'ancienne librairie **UnitGInt.pas** ou **UnitGInt.dll**, si certains concepts sont identiques comme le format GInt, il n'en va pas de même des fonctions et procédures qui ont été entièrement réécrites. Donc ces librairies ne sont pas compatibles.

Les nombres entiers naturels de l'ensemble \mathbb{N} s'étendent de 0 à l'infini en s'élevant de 1 en 1. Rien de plus simple depuis que l'Homo Sapiens compte. Avec Delphi en langage Pascal, les nombres entiers naturels sont définis par le type **longword** ou par le type **Int64**. Dans le premier cas, ils s'expriment sur une étendue binaire de 32 bits et sur 64 dans le second. Ceci est tributaire de la structure en 32 bits des registres de calculs des microprocesseurs. Le type Int64 n'étant lui qu'une possibilité offerte par la combinaison codée de deux registres de 32 bits. Donc, il découle de ce fait que la valeur d'un entier naturel exprimée en base décimale ne pourra excéder 4 294 967 295. Pour les petits calculs de la vie courante, c'est déjà bien grand et suffisant, certes, mais cette limite est vite franchie par de simples calculs de puissances ou de factorielles voire parfois par de simples multiplications. Embêtant ? Oui ou non... Au-delà rien n'interdit d'utiliser les types de nombres manipulés par les coprocesseurs mathématiques sous forme logarithmique. Oui, mais quid de la valeur exacte des nombres ? Et l'arithmétique modulaire ? Prenons l'exemple du cryptage des données qui s'appuie sur des nombres premiers suffisamment grands (plus de 512 bits) : impossible avec les types entiers de Delphi ! **NewGInt.pas** et/ou **NewGCent.Pas** constituent des tentatives particulières pour dépasser allègrement cette barrière du 32 bits pour les calculs sur des entiers naturels tout en préservant les habitudes de programmation du langage pascal. Elles constituent donc des bibliothèques de procédures et fonctions de base pour satisfaire aux besoins des opérations fondamentales sur des valeurs très grandes.

Genèse

La première idée de conception est de considérer que la numération quelle que soit la base utilisée se fait toujours au moyen de chiffres. Le terme le plus approprié serait sans doute **digits**. Car si l'humain depuis longtemps utilise une base d'écriture décimale avec 10 chiffres, il faut bien admettre que les processeurs eux se régalaient d'une base binaire à 2 chiffres. Mais, l'unité minimale de stockage informatique étant l'octet, c'est dans cette structure qu'il faut considérer les digits de la numération que l'on veut utiliser. Donc 1 octet = 1 digit. Tout dépend alors de la valeur maximale que l'on veut attribuer à chacun de ces octets-digits. Un octet de 8 bits peut avoir une valeur maximale de 255. Soit, mais rien n'interdit de la limiter par convention. Ainsi, si on se limite à 9, on sera dans une base strictement décimale, si on se limite à 99 on sera dans une numération en base 100 et si on ne se plie à aucune limite donc jusqu'à 255, on sera en base 256. Prenons quelques exemples d'écriture et de stockage pour mieux assimiler ces notions : imaginons que l'on veuille stocker la valeur 12563 exprimée en base 10 humaine. En digits-octets décimaux il faudra donc 5 octets à la suite : 1/2/5/6/3 chaque octet ne devant pas dépasser 9. En digits-octets base 100, il ne faudra plus que 3 octets : 1/25/63 chacun ne dépassant pas 99. En digits-octets base 256, il ne faudra plus que deux digits-octets : 49/19 (soit $49 \times 256 + 19 = 12563$).

Pour contenir et manipuler des collections d'octets en Delphi, deux solutions se présentent : soit des tableaux de valeurs (**array of...**) soit des suites d'octets en chaînes (**string**). Le type string semble d'évidence le plus proche des habitudes de numération humaine : des octets rangés en mémoire à la queue leu leu (aucun problème de codage little Indian), avec une adresse de départ et une longueur bien définies. De plus, pour le programmeur, toute une pléthore de procédures et fonctions est offerte par Delphi pour gérer et manipuler ces chaînes longues, pourquoi s'en priver ?

Donc les types **GInt** et **GCent** ne sont rien d'autres que des chaînes longues **AnsiString** du Pascal. Mais attention ! Leurs octets n'ont rien à voir avec des caractères alphanumériques ! Ce sont des valeurs numériques binaires accumulées en base 256 pour le GInt et en base 100 pour le GCent. Deux petits exemples pour mieux comprendre cela :

1/ GInt : soit une chaîne de cinq octets de long avec les valeurs successives 230, 136, 24, 156, 212. Cela représente, converti en base décimale : $(230 \times 256^4) + (136 \times 256^3) + (24 \times 256^2) + (156 \times 256^1) + (212 \times 256^0)$. Faites le calcul... Et imaginez la valeur décimale contenue dans un GInt de 12500 octets... de 560000 octets... Rêveur, ou rêveuse ?

2/ GCent : soit une chaîne de cinq octets avec les valeurs successives 95, 68, 23, 10, 69. Cela représente en base décimale : $(95 \times 100^4) + (68 \times 100^3) + (23 \times 100^2) + (10 \times 100^1) + (69 \times 100^0)$. Certes la structure contient beaucoup moins (2 fois et demi) ce que peut emmagasiner un GInt.

Conséquence directe, le GInt et le GCent de valeur nulle ont au moins un octet (#0) de long.

L'optimisation de la plupart des routines de calculs et conversions de NewGInt et NewGCent a été obtenue par un recours massif au langage ASM X86 intégré à Delphi et, ainsi, elles demeurent facilement compilables par les versions en 32 bits de ce langage. A chaque fois que cela a été possible, les algorithmes choisis pour les opérations de base ont résulté des meilleures performances aux tests de célérité pratiqués.

Pourquoi deux unités avec chacune un type bien particulier de stockage des grands entiers ? A priori, cela ne semble pas faire bon ménage ! D'un côté des processus calculatoires en base 256 et de l'autre des processus en base 100 totalement incompatibles pour les calculs. Mais, il n'a jamais été dit qu'il fallait obligatoirement les utiliser conjointement. Elles sont autonomes et présentent des opérations de base identiques. Selon les besoins du programmeur Delphi, celui-ci utilisera dans l'application qu'il développe l'une, ou l'autre, ou les deux à la fois. Chacune a son référentiel de calculs propre et d'emblée, les concepts ne sont absolument pas compatibles on l'a bien compris, mais les auteurs ont placé la possibilité de « passerelles » pour aller de l'une à l'autre dans le même programme, on verra comment plus loin.

Donc a priori deux bibliothèques bien distinctes mais qui peuvent devenir le cas échéant complémentaires. Si les deux ont été développées par les auteurs c'est que chacune possède ses avantages et ses inconvénients qui font que l'utilisation de l'une ou de l'autre dépend de l'analyse du programmeur en besoin d'une bibliothèque pour traiter des grands entiers.

➔ **GINT** : le gros avantage est que le format utilisé est dans la base native du processeur, cela garantit des calculs plus rapides sur des stockages en mémoire minimaux. Cette bibliothèque se prête aux développements algorithmiques binaires, et de ce fait offre des processus calculatoires supplémentaires comme les déplacements de bits et les opérations de logique booléenne. Cependant, pour concrétiser l'interfaçage avec la numération décimale humaine, les conversions base 10 \Leftrightarrow base 256 sont donc pénalisantes du point de vue vitesse d'exécution.

➔ **GCENT** : la base 100 donne une approche plus humaine des calculs. Elle permet le raisonnement sur les expressions des puissances de 10 et se révèle royale pour asseoir des applications de numérations autres que dans l'ensemble N. Le revers de la médaille est son besoin de mémoire plus important et une célérité des calculs très légèrement inférieure à celle des GInt (relativement car des cas sont surprenants). En revanche les conversions base 10 \Leftrightarrow base 100 sont pratiquement instantanées.

Dans un souci d'uniformisation sémantique, les interfaces des deux unités adoptent les mêmes principes lexicaux pour les noms des fonctions et procédures appelables. Ainsi, en principe général tout ce qui contient le vocable GInt relève de l'unité NewGINT et tout ce qui contient GCent relève de l'unité NewGCent. Pour les programmeurs Delphi, ils pourront remarquer que toute **function** commence par un **F**, toute **procedure** par un **P** et tout **test** par **Is**.

Fonctions et procédures de NewGINT.pas

A - Les convertisseurs

Les indispensables. Sans eux, impossible de parfaire une interface entre l'humain et les GInt. Ils ont cependant le défaut majeur d'être relativement lents (pour les nombres allant au-delà de plusieurs milliers de digits); il convient donc de ne pas les utiliser dans les phases de calculs binaires proprement dits, et de les réserver aux phases d'éditations ou d'affichages.

Function FStrToGInt(StrNb : string) : GInt ;

Function FGIntToStr(Nb : GInt) : string ;

Si on lui fournit en paramètre une chaîne ne contenant exclusivement que des caractères numériques, FStrToGInt renvoie le GInt correspondant. Elle fait une conversion de valeur base 10 \Rightarrow base 256. Attention, elle ne vérifie pas les caractères d'entrée et donne un résultat erroné si le domaine [0,..,9] des digits n'est pas respecté. Fonction inverse de la précédente, FGIntToStr convertit un nombre GInt en une chaîne affichable de caractères numériques. C'est donc une conversion base 256 \Rightarrow base 10.

Function FHexToGInt(StrHex: string) : GInt ;

Function FGIntToHex(Nb : string) : string ;

Il est tout aussi utile de pouvoir bâtir des programmes qui utilisent l'écriture hexadécimale des nombres plutôt que la base décimale, informatique mathématique oblige. Il faut donc là aussi les transformateurs adéquats. Comme ils convertissent des valeurs base hexadécimale \Leftrightarrow base 256, ils font appel à des calculs plus simples et sont donc plus rapides qu'en décimal. Mais toujours tenir compte que si les calculs sont assez rapides, ce sont les affichages des chaînes géantes qui réclament du temps d'exécution.

Function FBinToGInt(StrBin : AnsiString): GInt;

Function FGIntToBin(Nb: GInt): AnsiString;

Idem avec les expressions d'écritures binaires...

Function FIntToGInt(Nb: longword) : GInt ;

Function FInt64ToGInt(Nb : Int64) : GInt ;

Les entiers naturels de Delphi (nombres) n'étant pas compatibles avec les GInt (string), ils ont aussi besoin d'être convertis pour les calculs communs. Attention, les nombres entiers naturels ne sont jamais négatifs : tout integer sera systématiquement transtypé en longword non signé à la conversion.

Function FGIntToInt(Nb : GInt) : longword ;

D'utilisation peu fréquente, cette fonction convertit en longword les quatre derniers octets d'un GInt et ignore les autres si celui-ci est de longueur supérieure. A n'utiliser que si la certitude de GInt de 4 octets au plus est acquise.

Function FCopyGInt(Nb : GInt) : GInt;

Function FCopyLGInt(Nb : GInt ; Start,Count : longword) : GInt;

Chacune garantit que le GInt retourné sera bien une variable différente et non une instance de la même. Étant donné que certaines procédures peuvent influencer sur le contenu d'une variable GInt, il convient de pouvoir séparer les variables en RAM. Ainsi s'il est commode de faire **Nb2 :=Nb1** en Delphi, cela ne veut surtout pas dire que Nb1 et Nb2 sont deux variables distinctes donc **Nb2 :=CopyGInt(Nb1)** assoit en RAM deux variables bien distinctes.

La seconde fonctionne comme le Copy de Delphi. MAIS ! Et c'est très important, elle retourne un GInt et non pas une simple fraction de chaîne. Les zéros éventuels en tête sont éliminés.

B - Les générateurs

Ces fonctions sont destinées à générer des nombres GInt suivant le paramétrage corollaire.

Function FGINtNul : GInt ;

Renvoie directement un GInt égal à #0. C'est généralement une garantie de validité d'une nouvelle variable GInt, très utile avant les appels de procédures.

Function FRandomGInt(Nb : GInt) : GInt ;

Le GInt donné en paramètre définit le domaine de #0 < à <Nb dans lequel un nombre GInt aléatoire va être renvoyé.

Function FRandomLGInt(NbBits : longword) : GInt ;

Plus intéressante que la précédente, cette fonction génère un nombre GInt aléatoire possédant un nombre de bits compris entre les multiples de 8 immédiatement supérieur ou égal au paramètre.

Function FRandomLGIntImpair(NbBits : longword) : GInt ;

La même chose que la fonction précédente, mais tous les octets du GInt renvoyé sont impairs ; très utile si on veut une probabilité de nombre premier pour un éventuel test de primalité.

L'unité possède son propre générateur de pseudo-aléatoires ; inutile de faire précéder ces fonctions par des appels à Randomize de Delphi. Toutefois, si le programmeur le juge utile, il peut rafraîchir la variable de calcul par :

Procédure PRndGIntMize ;

Function FDeuxPowerN(Nb : longword) : GInt ;

Génératrice des GInt puissances de deux, 2^{Nb} , extrêmement rapide. En binaire, une puissance de deux est un bit 1 suivi par Nb zéros.

Function FMersenne(Nb : longword) : GInt ;

Génère le GInt nombre de Mersenne de rang Nb soit $2^{Nb} - 1$, elle aussi très rapide. En binaire, un nombre de Mersenne ne contient que des bits 1.

Function FPremier(Nb : longword) : GInt ;

Génère un nombre premier de Nb bits (Nb < 256). Il n'existe pas de fonction mathématique pour générer rapidement des nombres premiers. Basée sur le test de Miller-Rabbin, cette fonction est donc tout à fait aléatoire en temps d'exécution.

C - Les opérations élémentaires

Les GInt ayant été édités ou générés de la façon désirée, il s'agit maintenant de pouvoir réaliser sur et avec eux les opérations arithmétiques élémentaires. Des fonctions adéquates sont à disposition du programmeur. Dans un souci d'optimisation, on va trouver pour certaines opérations, la fonction et la procédure. Il s'agit en fait de recherches de gains de vitesses d'exécutions en fonction de cas particuliers. On retiendra que du fait qu'elles ignorent la recréation de chaînes de retour, les procédures sont nettement plus rapides que les fonctions, mais elles modifient la variable d'appel.

Function FAddGInt(Nb1, Nb2 : GInt) : GInt ;

Additionne les deux GInt passés en paramètres et renvoie le GInt résultant. Il s'agit d'une fonction qui retourne une variable GInt, elle préserve donc les deux GInt de départ. Il en sera toujours de même dans toutes les fonctions de calculs sur GInt.

Procédure PAddGInt(var Nb1 : GInt ; Nb2 : GInt) ;

Additionne directement la valeur de Nb2 à Nb1. Il va de soit que la variable Nb1 doit au préalable avoir une validité distincte.

Function FSubGInt(Nb1, Nb2 : GInt) : GInt ;

Chacun sait que la soustraction en entiers naturels ne peut se faire dans le sens petit moins grand. Donc cette fonction renverra systématiquement un GInt nul si Nb2>Nb1.

Procédure PSubGInt(var Nb1 : GInt ; Nb2 : GInt) ;

La même soustraction mais directement sur Nb1. On peut dire qu'il s'agit d'une diminution de celui-ci.

Function FDifGInt(Nb1, Nb2 : GInt) : GInt ;

Pour préserver l'avenir, c'est-à-dire une mutation des GInt vers les entiers relatifs, cette opération réalise systématiquement le plus grand moins le plus petit ; cela équivaut à calculer leur différence tout simplement.

Function FMulGInt(Nb1, Nb2 : GInt) : GInt ;

Multiplier deux GInt entre eux. Il est mathématiquement prouvé que l'algorithme naïf est le plus lent $O(n^2)$ en comparaison d'autres en calculs polynomiaux (Karatsuba). Malheureusement il s'est révélé aux essais que leur implémentation au-delà des 32 bits avec l'ASM octal devenait si gourmande en processus corollaires que le bon procédé naïf, oui, celui de l'école primaire, qui ne nécessite rien d'autre que des multiplications de digits-octets entre eux, en sortait vainqueur du point de vue célérité sur les très grands nombres GInt ! Mais cela n'a pas interdit d'adapter des processus plus rapides pour certains GInt particuliers.

Function FDivGInt(Nb1, Nb2 : GInt) : GInt ;

Ne nous fourvoyons pas, les divisions par 0 ou petit par grand se solderont par le retour d'un GInt nul (#0). Sinon, là, c'est l'algorithme dit de Hörner qui remporte la palme et haut la main, et c'est du très rapide même sur des nombres de plus de 10000 bits.

Function FModGInt(Nb1, Nb2 : GInt) : GInt ;

Sans cette opération, pas d'arithmétique dite modulaire. Elle renvoie le reste entier d'une division de deux GInt toujours possible sauf pour une division par 0 bien évidemment. Elle est aussi rapide que le calcul du quotient.

Procédure PDivModGInt(Nb1,Nb2 : GInt; var Q,R : GInt);

Quand on a besoin à la fois du quotient Q et du reste R d'une division de Nb1 par Nb2, pourquoi faire faire deux fois le même calcul ? Cette procédure renvoie les deux en variables prédéfinies deux fois plus vite.

Function FFactGInt(Nb : longword) : GInt ;

Function FFactorielleGInt(Nb : GInt) : GInt ;

Calcul de la factorielle d'un nombre entier. En Delphi 32 on ne va pas loin... Et en plus, les fonctions récursives saturent vite la pile système (stack overflow). Et bien avec les GInt ont va beaucoup... beaucoup plus loin. Oui mais, cela a été dit, la limite est technologique : 2^{32} octets, et puis il faut déjà partager l'espace requis, donc il faut être raisonnable. Le calcul de la factorielle d'un nombre de plus de 32 bits est trop long ! Et traînera une ribambelle de zéros. Les deux fonctions font la même chose, à la différence que la première veut un entier Delphi en paramètre alors que la seconde attend un GInt, mais celui-ci ne devra pas dépasser 4 octets de long.

Function FExpGInt(Nb : GInt, Exp : longword) : GInt ;

Function FPowerGInt(Nb, Exp : GInt) : GInt ;

Élévation à une puissance. Pour les très très grands nombres, ces calculs peuvent s'avérer relativement longs. Les deux fonctions élèvent un GInt Nb à la puissance Exp. Dans le premier cas, l'exposant est un nombre Delphi, dans le second, c'est un GInt. Attention, il ne faut quand même pas rêver ! Si le résultat attendu fait plusieurs dizaines de milliers de digits, l'ordinateur risque de s'attarder fortement, prévoir un lit à côté.

Function FPowerModGInt(Nb, Exp, Modulo : GInt) : GInt ;

Exponentiation modulaire rapide : cette fonction calcule $(Nb^{Exp}) \bmod Modulo$. Les adeptes de la cryptologie R.S.A. apprécieront certainement.

Function FCarreGInt(Nb : GInt) : GInt ;

Cette fonction a été conçue de manière à rendre le calcul Nb^2 beaucoup plus rapide que $FExpGInt(Nb,2)$. C'est tout. Mais son existence permettra à certains algorithmes sophistiqués de gagner en vitesse d'exécution. Trois fonctions depuis l'algorithme naïf à celui des carrés par impairs en passant par du Karatsuba...

Function FRacineGInt(Nb : GInt) : GInt ;

Si Nb est un carré parfait, cette fonction retourne le nombre entier équivalent à sa racine carrée, sinon celle-ci est calculée par défaut : il s'agira du nombre entier inférieur le plus proche. Très belle fonction qui s'appuie sur le célèbre algorithme dit du « goutte-à-goutte ».

Function FPGCDGInt(Nb1, Nb2 : GInt) : GInt ;

Function FPPCMGInt(Nb1, Nb2 : GInt) : GInt ;

Oui, on ne rêve pas. Il s'agit du calcul du P.G.C.D. de deux grands nombres, parfois si utile. Dédit par l'algorithme d'Euclide, il s'obtient aussi très vite. Le P.P.C.M. est lui aussi vite induit par le premier.

C - Les opérations utilitaires très rapides

Il aurait été vraiment mesquin de ne s'arrêter qu'aux opérations arithmétiques élémentaires décrites ci-dessus alors que les apports spécifiques de l'arithmétique binaire n'eussent pas été exploités pour des calculs, spéciaux certes, mais d'utilité très courante. Ultra-rapides parce que ne faisant appel qu'à des opérations élémentaires du processeur, ne surtout pas négliger d'y avoir recours au besoin.

Function FIncGInt(Nb : GInt) : GInt ;

Procédure PIncGInt(var Nb : GInt) ;

Function FDecGInt(Nb : GInt) : GInt ;

Procédure PDecGInt(var Nb : GInt) ;

Et oui ! Les boucles et itérations diverses. Attention les fonctions renvoient bien un GInt incrémenté ou décrémenté, mais différent de celui d'origine qui demeure. Les procédures, elles, incrémentent ou décrémentent la variable elle-même qui doit donc avoir son existence prédéfinie. Ainsi **Nb2 :=FIncGInt(Nb1)** renvoie $Nb2=Nb1+1$ alors que **PIncGInt(Nb1)** incrémente la valeur contenue dans Nb1. A noter que les procédures qui n'ont pas à recopier de chaînes, sauf en cas de dépassements, sont nettement plus rapides que les fonctions.

D - Les décalages

Function FShIGIntOctets(Nb : GInt, N : longword) : GInt ;

Multiplie un GInt par N fois 256. Ce qui revient à décaler les octets de N fois vers la gauche.

Function FShrGIntOctets(Nb : GInt, N : longword) : GInt ;

Divise un GInt par N fois 256. Ce qui revient à décaler les octets de N fois vers la droite.

Function FShIGIntBits(Nb : GInt, N : longword) : GInt ;

Procédure PShIGIntBits(var Nb : GInt, N : longword) ;

Multiplie un GInt par N fois 2. Ce qui revient à décaler les bits de N fois vers la gauche.

Function FShrGIntBit(Nb : GInt, N : longword) : GInt ;

Procédure PShrGIntBit(var Nb : GInt, N : longword) ;

Divise un GInt par N fois 2. Ce qui revient à décaler les bits de N fois vers la droite.

E - Les opérations logiques

Function FAndGInt(Nb1,Nb2 : GInt): GInt;

Effectue un AND logique bit à bit entre les deux GInt.

Function FOrGInt(Nb1,Nb2 : GInt): GInt;

Effectue un OR logique bit à bit entre les deux GInt.

Function FXorGInt(Nb1,Nb2 : GInt): GInt;

Effectue un XOR logique bit à bit entre les deux GInt.

F - Les tests

Function IsCompGInt(Nb1,Nb2 : GInt): integer;

Comparer deux GInt, pourquoi une fonction spéciale puisque s'agissant de strings, Delphi possède déjà ses propres outils ? Il nous a semblé utile d'implémenter une fonction dont on sait qu'elle ne sera pas rédhibitoire en microsecondes dans les itérations nombreuses. Elle retourne un integer et pas un boolean : Nb1>Nb2 si >0 // Nb1=Nb2 si =0 // Nb1<Nb2 si <0.

Function IsGIntNul(Nb : GInt): boolean;

La belle affaire! Renvoie true si Nb vaut #0. Très utile dans les tests de fin d'itérations. Elle est l'équivalente de **if Nb = #0** en Delphi.

Function IsGIntFull(Nb : GInt): boolean;

Elle aurait pu rester en fonction interne de l'unité. Mais il s'est avéré que certains en ont eu besoin. Elle renvoie true si **tous** les octets du Nb sont pleins, c'est-à-dire tous égaux à 255 ou \$FF.

Function IsGIntPair(Nb : GInt): boolean;

True si Nb est un nombre pair et false si impair.

Function IsGIntPrime(Nb : GInt): boolean;

Vous avez dit RSA ? Indispensable, ce test probabiliste (Miller-Rabbin) qui renvoie true si le GInt est très probablement premier à 99,999% et false s'il ne l'est pas à 100%.

Function IsGIntCarre(Nb : GInt): boolean;

Renvoie true si le GInt passé en argument est un carré parfait. Ne rêvons pas, cette fonction ne s'affranchit nullement de passer par le calcul de la racine carrée ré-élevée au carré ! Elle n'est donc pas très rapide, surtout sur les très grands nombres.

Procedure IsDiv2GInt(Nb: GInt; var P2: boolean) : longword ;

Doublement utile, elle teste si le GInt est une puissance de 2, répond par le paramètre **PE2**, mais calcule et renvoie le nombre de 0 présents à la fin du GInt, c'est-à-dire le degré de divisibilité par 2.

Procedure IsLogDeux(Nb: GInt) : longword ;

Renvoie le logarithme binaire du GInt, qui est égal à l'exposant de la puissance de 2 la plus grande et inférieure ou égale à Nb .

Fonctions et procédures de NewGCENT.pas

A - Les convertisseurs

Function **FStrToGCent**(StrNb : string) : GCent;

Function **FGCentToStr**(Nb : GCent) : string ;

Si on lui fournit en paramètre une chaîne ne contenant exclusivement que des caractères numériques, FStrToGCent renvoie le GCent correspondant. Elle fait une conversion de valeur base 10 \Rightarrow base 100. Attention, elle ne vérifie pas les caractères d'entrée et donne un résultat erroné si le domaine [0,..,9] des digits n'est pas respecté. Fonction inverse de la précédente, FGCentToStr convertit un nombre GCent en une chaîne affichable de caractères numériques. C'est donc une conversion base 100 \Rightarrow base 10. Très rapide.

Function **FlntToGCent**(Nb: longword) : GCent ;

Function **Flnt64ToGCent**(Nb : Int64) : GCent ;

Les entiers naturels de Delphi (nombres) n'étant pas compatibles avec les GCent (string), ils ont aussi besoin d'être convertis pour les calculs communs. Attention, les nombres entiers naturels ne sont jamais négatifs : tout integer sera systématiquement transtypé en longword non signé à la conversion.

Function **FGCentToInt**(Nb : GCent) : longword ;

Function **FGCentToInt64**(Nb : GCent) : Int64 ;

D'utilisation peu fréquente, ces fonctions convertissent en longword les cinq derniers octets d'un GCent et en Int64 les 10 derniers octets d'un GCent ignorent les autres si celui-ci est de longueur supérieure. A n'utiliser que si la certitude de GCent de 5 ou 10 octets au plus est acquise.

Function **FCopyGCent**(Nb : GCent) : GCent;

Function **FCopyLGCent**(Nb : GCent ; Start,Count : longword) : GCent;

Garantit que le GCent retourné sera bien une variable différente et non une instance de la même. Étant donné que certaines procédures peuvent influencer sur le contenu d'une variable GCent, il convient de pouvoir séparer les variables en RAM. Ainsi s'il est commode de faire **Nb2 :=Nb1** en Delphi, cela ne veut surtout pas dire que Nb1 et Nb2 sont deux variables distinctes donc **Nb2 :=CopyGCent(Nb1)** assoit en RAM deux variables bien distinctes.

La seconde fonctionne comme le Copy de Delphi, mais garantit que le retour est bien un GCent et pas une fraction de chaîne avec éventuellement des zéros en tête.

Function **FGCentToGInt**(Nb : GCent) : GInt;

Function **FGIntToGCent**(Nb : GInt) : GCent;

Il faut rendre à César ce qui appartient à César ! Il n'est pas possible de faire cohabiter du GCent avec du GInt dans les calculs. Ou on est dans l'une des bibliothèques ou dans l'autre. Cependant, des routines de calculs existent dans l'une et pas dans l'autre. Dommage ! Et s'il était possible de convertir un GInt en GCent et vice versa pour pouvoir s'en servir quand même ? Ces deux fonctions sont les passerelles entre les deux bibliothèques.

- Exemple 1 : j'utilise la bibliothèque des GCent et je veux faire un traitement binaire ? Pas de problème, je convertis mon GCent en GInt, je fais le traitement et je reconvertis mon GInt en GCent.

- Exemple 2 : j'utilise la bibliothèque des GInt et je veux faire subir un traitement décimal qui n'existe que dans la bibliothèque des GCent ? Même démarche mais en sens inverse.

B - Les générateurs

Ces fonctions sont destinées à générer des nombres GCent suivant le paramétrage corollaire.

Function FGCentNul : GCent ;

Renvoie directement un GCent égal à #0. C'est généralement une garantie de validité d'une nouvelle variable GCent, très utile avant les appels de procédures.

Function FRandomGCent(NbDigits : Longword ; const odd : boolean) : GCent ;

Retourne un GCent aléatoire de NbDigits de long. Si la constante Odd est à true, il sera pair, ou impair si elle est à false. L'unité possède son propre générateur de pseudo-aléatoires. Cependant, s'il le juge utile, le programmeur peut régénérer la semence de ce générateur par :

Procedure PRndGCentMize ;

Function FDixPowerN(Nb : longword) : GCent ;

Génératrice des GCent puissances de dix, 10^{Nb} , extrêmement rapide. En décimal, une puissance de dix est un digit 1 suivi par Nb zéros.

C - Les opérations élémentaires

Dans un souci d'optimisation, on va trouver pour certaines opérations, la fonction et la procédure. Il s'agit en fait de recherches de gains de vitesses d'exécutions en fonction de cas particuliers. On retiendra que du fait qu'elles ignorent la recréation de chaînes de retour, les procédures sont nettement plus rapides que les fonctions, mais elles modifient la variable d'appel.

Function FAddGCent(Nb1, Nb2 : GCent) : GCent ;

Procedure PAddGCent(var Nb1 : GCent ; Nb2 : GCent) ;

Additionne les deux GCent passés en paramètres. La fonction qui retourne une variable GCent préserve donc les deux GCent de départ. La procédure modifie la première variable tout en allant nettement plus vite. Il en sera toujours de même dans toutes les fonctions et procédures de calculs sur GCent.

Function FSubGCent(Nb1, Nb2 : GCent) : GCent ;

Procedure PSubGCent(var Nb1 : GCent ; Nb2 : GCent) ;

Function FDifGCent(Nb1, Nb2 : GCent) : GCent ;

Dans les trois cas on soustrait un GCent d'un autre. Les deux premières s'arrêteront à #0, pas la dernière qui soustrait le petit du grand.

Function FMulGCent(Nb1, Nb2 : GCent) : GCent ;

Multiplier deux GCent entre eux.

Procedure PDivModGCent(Nb1, Nb2 : GCent; var Q, R : GCent);

Cette procédure renvoie à la fois le quotient Q et le reste R de la division de Nb1 par Nb2.

Function FExpGCent(Nb : GCent ; Exp : longword) : GCent ;

Function FPuissanceGCent(Nb : GCent ; Exp : longword) : GCent ;

Il s'agit dans les deux cas d'élever le GCent Nb à la puissance Exp. La différence tient dans le fait que la première est itérative et la seconde récursive.

Function **FCarreGCent**(Nb : GCent) : GCent ;

Function **FRacineGCent**(Nb : GCent) : GCent ;

Soit Nb un nombre GCent, la première en renvoie son carré alors que la seconde retourne sa racine carrée entière par défaut.

C - Les opérations utilitaires rapides

Function **FIncGCent**(Nb : GCent) : GCent ;

Procédure **PIncGCent**(var Nb : GCent) ;

Function **FDecGCent**(Nb : GCent) : GCent ;

Procédure **PDecGCent**(var Nb : GCent) ;

Attention les fonctions renvoient bien un GCent incrémenté ou décrémenté, mais différent de celui d'origine qui demeure. Les procédures, elles, incrémentent ou décrémentent la variable elle-même qui doit donc avoir son existence prédéfinie. Ainsi **Nb2 :=FIncGCent(Nb1)** renvoie Nb2=Nb1+1 alors que **PIncGCent(Nb1)** incrémente la valeur contenue dans Nb1. A noter que les procédures qui n'ont pas à recopier de chaînes, sauf en cas de dépassements, sont nettement plus rapides que les fonctions.

Function **FDemiGCent**(Nb : GCent) : GCent ;

Procédure **PDemiGCent**(var Nb : GCent) ;

Function **FDoubleGCent**(Nb : GCent) : GCent ;

Procédure **PDoubleGCent**(var Nb : GCent) ;

Les GCent n'étant pas en base binaire, les fonctions de double et moitié ne se traduisent pas par de simples décalages d'un bit. Il faut donc bien avoir ces calculs directs à disposition.

D - Les décalages

Function **FShINGCent**(Nb : GCent, N : longword) : GCent ;

Multiplie un GCent par N fois 100. Ce qui revient à décaler les octets de N fois vers la gauche.

Function **FShrNGCent**(Nb : GCent, N : longword) : GCent ;

Divise un GCent par N fois 100. Ce qui revient à décaler les octets de N fois vers la droite.

Function **FGCentMulN10**(Nb : GCent, N : longword) : GCent ;

Multiplie un GCent par N fois 10.

Function **FGCentDivN10**(Nb : GCent, N : longword) : GCent ;

Divise un GCent par N fois 10.

E - Les tests

Function **IsCompGCent(Nb1,Nb2 : GCent): shortInt;**

Comparer deux GCent, pourquoi une fonction spéciale puisque s'agissant de strings, Delphi possède déjà ses propres outils ? Il nous a semblé utile d'implémenter une fonction dont on sait qu'elle ne sera pas rédhibitoire en microsecondes dans les itérations nombreuses. Elle retourne : +1 si Nb1>Nb2 // 0 si Nb1=Nb2 // -1 si Nb1<Nb2.

Function **IsGCentNul(Nb : GCent): boolean;**

La belle affaire! Renvoie true si Nb vaut #0. Très utile dans les tests de fin d'itérations. Elle est l'équivalente de **if Nb = #0** en Delphi.

Function **IsGCentFull (Nb : GCent): boolean;**

Elle renvoie true si **tous** les octets du Nb sont pleins, c'est-à-dire tous égaux à 99.

Function IsGCentPair(Nb : GCent): boolean;
True si Nb est un nombre pair et false si impair.

Function IsGCentMult3(Nb : GCent): boolean;
Renvoie true si le GCent est un multiple de 3.

Function IsGCentMult5(Nb : GCent): boolean;
Renvoie true si le GCent est un multiple de 5.

Astuces et conseils communs

1/ Comment intégrer ces bibliothèques de routines dans un programme Delphi ? Rien de plus simple, on place NewGInt ou NewGCent ou les deux à la fois dans les uses du programme, on développe et on compile.

2/ Nous sommes tentés d'affirmer que pour tout programmeur en Delphi, confirmé ou non, aucun conseil particulier n'est nécessaire pour utiliser les routines de ces unités dans ses programmes. Mais tout de même, la manipulation d'énormes quantités de données (d'octets) demande un minimum de précautions. Certes, Delphi sait, ô combien, gérer ses strings en toute transparence, mais l'utilisation de gros GInt ou GCent et des fonctions de calculs adéquates entraîne des appels à d'énormes buffers dits de passage en mémoire. Ceux-ci sont à chaque étape, calculés au plus juste, mais quid de la quantité de RAM disponible gérée par Delphi si on vient à enchâsser des appels de fonctions gourmandes ? Bien prendre soin d'écrire les étapes de calculs en les isolant les unes des autres garantit de ne pas « empiler » des zones de mémoire indisponibles. Un exemple pour mieux comprendre :

A, B, C, D sont des variables GInt. Imaginons que l'on veuille effectuer le calcul suivant : $D = (A+B) \times C \text{ div } 2$. Traduction en Delphi avec les fonctions de NewGInt :

```
D := FShrGIntBits(FMulGInt(C,FAddGInt(A,B)),1);
```

Ça passe sûrement... Mais s'il s'agit de nombres avec plusieurs milliers de digits décimaux chacun, on risque une bonne violation d'accès pour cause de bousculade de buffers en mémoire ou un net ralentissement des calculs. Alors soyons plus pragmatiques ou moins pressés dans nos écritures. Au diable la concision du Pascal si elle se révèle un pis-aller :

```
D := FAddGInt(A,B) ;  
D := FMulGInt(C,D) ;  
PShrGIntBits(D,1) ;
```

Finalement c'est plus « lisible » pour tout le monde, cela garantit une libération des buffers de passage à chaque étape du déroulement du code et c'est certainement plus rapide.

3/ Tout a été pensé (osons l'espérer) pour que les phases de calculs soient les plus rapides possible. Mais, rançon de la gloire, dans un interfaçage, il faudra bien passer par un affichage « lisible » des résultats. Pour le téméraire qui voudrait « voir » des quantités dépassant des multiples de gogols, il faudra quand-même être patient... Mais si des itérations de calculs sur nombres géants doivent avoir lieu : proscrire les conversions et les affichages dans les boucles. Malgré tout, et si on veut bien rester humble, la vitesse d'exécution est tout à fait appréciable. Pour s'en convaincre, il vous suffit d'essayer ce petit exemple :

*On demande à Delphi de générer un nouveau projet avec un composant **TForm** sur lequel on dépose un composant **TButton1** et un composant **TMemo1** que l'on dispose afin de ménager une assez grande surface d'affichage. On place **NewGCent** dans les **Uses**, puis dans l'événement **OnClick** du bouton, on écrit ce petit programme :*

```

procedure TForm1.Button1Click(Sender: TObject);
var N,S,T: GCent; Start : Int64;
begin
    Start:= GetTickCount;
    Randomize;
    N:=FRandomGCent(202,false); S:=FDixPowerN(132);
    memo1.Lines.Add(FGCentToStr(N)); memo1.Lines.Add(' ');
    memo1.Lines.Add(FGCentToStr(S)); memo1.Lines.Add(' ');
    T:=FMulGCent(N,S);
    memo1.Lines.Add(FGCentToStr(T)); memo1.Lines.Add(' ');
    T :=FDemiGCent(T);
    memo1.Lines.Add(FGCentToStr(T)); memo1.Lines.Add(' ');
    T:=FExpGCent(T,12);
    memo1.Lines.Add(FGCentToStr(T)); memo1.Lines.Add(' ');
    ShowMessage ('Mis en tout : '+IntToStr(GetTickCount - Start)+' millisecondes, conversions et affichages inclus');
end;

```

Puis lancez le programme et cliquez sur le bouton.... Avez-vous eu le temps de chronométrer ? Et pourtant : génération d'un nombre aléatoire de 202 digits, puis de 10^{132} , que l'on multiplie entre eux, que l'on divise par 2 puis qu'on élève à la puissance 12 !

Conclusion

L'assembleur de Delphi est un outil très précieux pour qui sait l'utiliser. Il est vrai que peu de programmeurs ont le temps et la patience de re-développer ainsi leurs routines de base. Folie ? C'est parce qu'il voulait se consacrer à la science des codes et de la cryptologie que l'auteur de cette unité, adepte de Delphi et de son assembleur intégré, n'ayant pas trouvé d'outils tout faits en pascal (on trouve ici et là sur le Net des bibliothèques écrites en C), s'est résolu à les créer lui-même. Deux années d'élucubrations, travaux, recherches et essais. Il espère donc que ces bibliothèques donneront satisfaction à quelques passionnés de l'informatique avec Delphi pour développer leurs projets personnels. Il met toutefois l'utilisateur potentiel en garde contre une surenchère de résultats escomptés. Ce n'est pas parce que NewGINT et NewGCENT permettent des calculs astronomiques qu'il faille tenter de leur demander de l'astronomie « givré » et irréaliste pour ainsi affirmer qu'il y a des failles dans ce bel ensemble. Bien sûr qu'il y en a : taper sur de l'exponentiel à tout schuss est le meilleur moyen de « décoller » ou de s'endormir devant la machine qui ronronne !

*L'auteur, **Kinzinger René**, remercie particulièrement **Geyer Gilbert** pour son aide précieuse, ses solutions de tests performants et ses suggestions d'amélioration des processus de calculs.*

Perspectives

Répetons-le ces unités, ouvertes et mises à disposition de tous les développeurs Delphi, ne sont que des bibliothèques de routines de base. Rien n'interdit à tout un chacun de les utiliser asservies aux besoins et envies qu'il veut. Après tout, on détourne bien la recette du pain pour en faire de la fougasse ou de la pizza ! Voici à titre d'exemples deux idées de développements particuliers qui utilisent ces routines de calculs.

1 – Des ZInt par Kr85 :

A part le signe, rien ne différencie un nombre entier naturel (\mathbb{N}) d'un nombre entier relatif (\mathbb{Z}).

Type **ZInt = record**

 Signe : boolean ;

 Valeur : GInt ;

End ;

Et puis d'établir des routines de conversions et calculs appropriées ? Par exemple :

```
Function FMulZInt(Nb1,Nb2 : ZInt) : ZInt ;  
Begin  
  Result.Signes := (Nb1.Signes=Nb2.Signes);  
  Result.valeur :=FMulGInt (Nb1.Valeur,Nb2.Valeur) ;  
End;
```

```
Function FZIntToStr(Nb : ZInt) : string ;  
Begin  
  Case Nb.Signes of  
    True : Result:=' ' + FGCentToStr(Nb.Valeur);  
    False: Result:='- ' + FGCentToStr(Nb.Valeur);  
  End;  
End;
```

Etc, etc ...

2 – Des décimaux dans \mathbb{R} par Gilbert Geyer :

Bien que les unités aient été conçues pour manipuler des entiers naturels géants, leurs fonctions de base sont également utilisables pour des applications où il est souhaité de repousser la barrière des 19 à 20 décimales de Delphi donc pour du calcul algébrique dans le domaine des nombres réels. Il suffit à cet effet de gérer dans l'applicatif la manipulation de strings numériques en base 10 ainsi que le signe et les puissances de dix (qui donnent soit la position de la virgule dans une string représentative d'une valeur entière, soit le nombre de zéros à ajouter à droite) puis de faire appel aux fonctions de NewGCent avec des valeurs entières corrigées d'un nombre de puissances de dix approprié au résultat escompté puis de positionner la virgule au bon endroit dans la valeur entière du résultat du calcul. On peut ainsi facilement obtenir par exemple la racine carrée de 2 avec un nombre « géant » de décimales ... ou faire des calculs avec des nombres réels hyper-microscopiques (suffit d'utiliser un Int64 négatif pour gérer l'exposant des puissances de dix).

En bref les GInt et les GCent permettent de couvrir l'étendue depuis les nombres géants aux nombres lilliputiens !