Utilisation de GNAT GPL

L'utilisation de GNAT est décrite dans deux manuels :

- le manuel utilisateur "GNAT GPL User's Guide" décrit les caractéristiques du compilateur et des utilitaires nécessaires pour construire un programme Ada.
- le manuel de déférence "GNAT Reference Manual" contient les informations nécessaire pour construire des programmes Ada notamment les caractéristiques Ada dépendantes du compilateur ainsi que les bibliothèques spécifiques à GNAT.

Ils sont accessible sur :

www.adacore.com/developers/documentation

et en local sur :

/usr/local/gnat/share/doc/gnat

Voir l'installation du compilateur GNAT sur Blady.

Sommaire

1. Utilisation avec le Terminal

La commande "gnatmake" seule, sans paramètre, donne justement la liste des paramètres possibles. Néanmoins, la simple commande suivante donnera de bons résultats :

```
$ gnatmake hello.adb
```

Le fichier hello.adb étant :

```
with Text_IO; use Text_IO;
procedure Hello is
begin
put_line("Hello again, avec Ada.");
end;
```

Et les résultats ne se font pas attendre :

```
$ gnatmake hello.adb
gcc -c hello.adb
gnatbind -x hello.ali
gnatlink hello.ali
$ ./hello
Hello again, avec Ada.
```

2. Les commandes utiles avec le Terminal

La liste des commandes est obtenue de la façon suivante :

```
$ gnat
GNAT Community 2020 (20200429-84)
Copyright 1996-2020, Free Software Foundation, Inc.

To list Ada build switches use --help-ada

List of available commands

GNAT BIND               gnatbind
réalise l'édition des liens Ada des unités compilées
GNAT CHOP               gnatchop
découpe un fichier en unités pour satisfaire les conventions gnat
GNAT CLEAN              gnatclean
nettoie les fichiers générés par gnat (non présent avec gnat-osx)
GNAT COMPILE            gnatmake -f -u -c
compile une entité Ada
GNAT CHECK              gnatcheck
vérifie le code source suivant des règles définies (non présent avec gnat-osx)
GNAT ELIM               gnatelim
détecte et élimine les sous-programmes inutilisés (non présent avec gnat-osx)
GNAT FIND               gnatfind
liste toutes les utilisations d'une entité Ada
GNAT KRUNCH             gnatkr
réduit les noms de fichiers au nombre maximal de lettres spécifié
```

GNAT LINK gnatlink
réalise l'édition des liens de l'exécutable sur l'OS cible
GNAT LIST gnatls
liste le contenu des objets générés
GNAT MAKE gnatmake
utilitaire optimisé de compilation multi-unités
GNAT METRIC gnatmetric
statistiques sur le code Ada (non présent avec gnat-osx)
GNAT NAME gnatname
réalise la correspondance entre les unités Ada et les noms des fichiers lorsque ceux-ci ne sont pas au standard gnat
GNAT PREPROCESS gnatprep
pré-processeur externe
GNAT PRETTY gnatpp
reformate le source Ada (non présent avec gnat-osx)
GNAT STACK gnatstack
calcul la taille de pile mémoire maximale théorique (non présent avec gnat-osx et gnat-gpl)
GNAT STUB gnatstub
créé le squelette d'un corps à partir d'une spécification (non présent avec gnat-osx)
GNAT TEST gnattest
créé le squelette des tests du type AUnit (non présent avec gnat-osx)
GNAT XREF gnatxref
utilitaire d'édition des références croisées

De même chacune des commandes exécutée sans argument affichera justement la liste des arguments possibles.

```
$ gnatmake (extrait)
Usage: gnatmake  opts  name  {[-cargs opts] [-bargs opts] [-largs opts]}
  name  is a file name from which you can omit the .adb or .ads suffix
gnatmake switches:
  --version   Display version and exit
  --help      Display usage and exit
  -c       Compile only
  -f       Force recompilations of non predefined units
  -k       Keep going after compilation errors
  -m       Minimal recompilation
  -M       List object file dependences for Makefile
  -n       Check objects up to date, output next file to compile if not
  -o name  Choose an alternate executable name
  -p       Create missing obj, lib and exec dirs
  -Pproj   Use GNAT Project File proj
  -s       Recompile if compiler switches have changed
  -u       Unique compilation. Only compile the given file.
  -v       Display reasons for all (re)compilations
  -z       No main subprogram (zero main)
  --keep-temp-files   Keep temporary files
To pass an arbitrary switch to the Compiler, Binder or Linker:
  -cargs opts   opts are passed to the compiler
  -bargs opts   opts are passed to the binder
  -largs opts   opts are passed to the linker
Compiler switches (passed to the compiler by gnatmake):
  -Idir        Specify source files search path
 -gnat2012 Ada 2012 mode (default)
  -gnat-p   Cancel effect of previous -gnatp switch
```

Et aussi avec gcc :
$ gcc --help

3. Convention des noms de fichiers avec GNAT

Par défaut, le nom du fichier doit être identique à celui de l'unité Ada définie dans le source. Ne pas mettre d'espaces ou des tirets réservés aux unités filles. Un nom de fichier doit toujours figuré en caractères minuscules.

'main.ads'
Main (spec), contient la spécification d'une procédure ou d'une fonction, utile pour l'appeler d'une autre unité.

'main.adb'
Main (body), contient le code de la procédure ou de la fonction principale du programme.

'arith_functions.ads'
Arith_Functions (package ou proc ou func spec), contient les spécifications d'une unité Ada.

'arith_functions.adb'
Arith_Functions (package, proc ou func body), contient le code d'une unité Ada.

'func-spec.ads'
Func.Spec (child package spec), contient les spécifications d'une unité fille Ada.

'func-spec.adb'
Func.Spec (child package body), contient le code d'une unité fille Ada.

'main-sub.adb'
Sub (subunit of Main separate), contient le code d'une unité Ada définie comme séparée.

4. Les options utiles pour la compilation

La commande de compilation issue d'une commande en ligne ou de l'utilitaire de compilation gnatmake est en standard : **gcc -c essai.adb** (respectivement le compilateur, l'option de compilation et un source Ada).
GNAT comporte quelques options qui nous viennent en aide suivant les différentes phases de développement de notre programme :
- mise au point source (maximum d'information à la compilation),
- mise au point (maximum d'information à l'exécution),
- inspection du code et des données, pour la livraison de l'exécutable (maximum de performance à l'exécution).

Pour avoir l'ensemble des informations de vérifications de compilation du code source lors des premières compilations, j'utilise les options : **-gnatc -gnatf -gnatU -gnatw.eH.YD -Wall**

(respectivement : pas de génération de code, l'ensemble des erreurs, l'affichage avec l'étiquette "error" et l'ensemble des avertissements sauf trois voir §8.d ainsi que les avertissements du générateur de code) ou plus simplement

-gnatcfUw.eH.YD -Wall (l'option c doit être la première et l'option w la dernière).

À ce stade la génération du code exécutable n'est pas nécessaire.

Pour obtenir le listing du code source avec les erreurs, on peut utiliser **-gnatv** ou **-gnatl** pour le listing complet, mais aujourd'hui avec les environnements de développement intégré comme GPS, Xcode ou jGrasp ce n'est pas nécessaire.

Pour afficher la liste de l'ensemble des unités requises pour la compilation, on peut utiliser l'option **-gnatu**.

Lorsque tout le code source compile, pour obtenir l'ensemble des vérifications d'exécution du programme et des informations de déverminage, j'utilise les options : **-fstack-check -g -gnato -gnatVaep -gnatE -gnateA -gnateE -gnateF -gnateV -bargs -Es**

(respectivement le contrôle de la pile, la génération des information pour le dévermineur, le contrôle des dépassements numériques (en général et pour les assertions, nécessaire pour lever les exceptions numériques comme Dynamic_predicate en Ada 2012), le contrôle de la validité des valeurs avant l'utilisation, le contrôle dynamique des élaborations de code, pas de lien entre deux paramètres du même sous-programme, l'affichage d'informations supplémentaires pour les exceptions, les dépassements numériques pour les réels non contraints, la validité des paramètres d'un sous-programme et de la pile des appels symboliques dans les messages des exceptions - note : sur macOS on n'obtient que les adresses, les appels symboliques sont fournis par l'utilitaire atos, voir les exceptions sur Blady).

À ce stade la génération d'un exécutable optimisé n'est pas nécessaire.

Pour activer les pragmas Assert, Debug ainsi que les aspects Ada 2012 (pré-conditions, post-conditions, invariance et prédicat de types), on utilisera l'option **-gnata**.

Pour traquer les variables sans valeurs initiales, le prama Initialize_Scalars va initialiser ces variables avec une valeur hors bornes ce qui va provoquer une exception lors d'une utilisation avant qu'elles ne soient initialisées. Pour une portée sur l'ensemble des codes sources, le pragma est mis dans un fichier de configuration de pragma puis référencé par l'option -gnatec=<fichier>.

Pour obtenir les instructions assembleurs générées, on utilisera les options **-fverbose-asm -S**.

Pour obtenir des informations détaillées sur la structure des données on peut utiliser l'option **-gnatR2s**.

Pour obtenir des informations détaillées sur l'unité Standard on peut utiliser l'option **-gnatS** :

```
$ gcc -c -gnatc -gnatS essai.adb
```

Lorsque tout fonctionne, pour obtenir un exécutable optimisé, j'utilise les options : **-gnatp -gnatn -O2**

(respectivement la suppression de tous les contrôles, la mise du code en ligne pour les procédures désignées par le pragma inline et l'optimisation étendue de gcc).

Pour aller plus loin dans l'optimisation, on peut utiliser l'option -O3 qui produit en plus la mise en ligne automatique du code des "petites" procédures (à la discrétion du compilateur). Ce qui n'est pas forcément intéressant sur des programmes avec beaucoup de code.

Bien sûr, il est possible d'obtenir tout d'un coup vérifications et optimisations avec les options :

-gnatEafUnoeAeEeFeVw.eH.YD -Wall -gnatVa -fstack-check -g -O2 -bargs -E, au risque de mélanger la nature des problèmes qui vont surgir et donc à rendre plus complexe leur résolution.

D'une part, il est étrange de vouloir faire des vérifications tout en optimisant le code…

On peut prendre alors les options -gnatfUnpw.eH.YD -Wall -g -O2 au dépend de la sécurité.

D'autre part, les environnements de développement intégré comme GPS permette de définir facilement plusieurs cibles avec des options différentes. Les principales options préconisées ci-dessus sont regroupées dans un exemple de fichier projet GPR sur github.com/Blady-Com/GPR_Template/blob/master/hello_prog.gpr avec les cibles "Profiling", "Optimizing", "Debugging".

5. Utilisation de bibliothèques utilisateurs

Par défaut, GNAT ne connaît que les unités de compilation de la bibliothèque standard Ada et la bibliothèque GNAT. L'emplacement de ces unités de compilation est situé dans les dossiers 'adainclude' et 'adalib' venant avec le compilateur. Lors de l'utilisation répétée d'unités de compilation utilisateurs, il est pratique d'indiquer au compilateur le chemin de ces unités une fois pour toute et non pas à chaque compilation (avec l'option -I).

Le compilateur GNAT a besoin de trois sortes de fichiers : les spécifications (fichiers ads), les dépendances entre unités (fichiers ali) et les binaires compilés (fichiers o, a, so ou dylib) de la bibliothèque utilisée.

a) Les chemins d'accès de la bibliothèque

La variable d'environnement ADA_INCLUDE_PATH donne le chemin d'accès des spécifications et ADA_OBJECTS_PATH celui des dépendances et binaires, par exemple :

```
$ export ADA_INCLUDE_PATH=/usr/local/Bindings/Frameworks/include:/usr/local/Bindings/ncurses/include
$ export ADA_OBJECTS_PATH=/usr/local/Bindings/Frameworks/lib:/usr/local/Bindings/ncurses/lib
```

b) Le fichier projet GPR

GNAT possède une possibilité d'utilisation de bibliothèque plus puissante avec les fichiers projets (.gpr). Ce fichier contient les informations d'accès aux fichiers de la bibliothèque. Le fichier projet de notre programme doit alors comporter la clause d'importation de la bibliothèque à son début, par exemple :

```
with "my_ada_lib.gpr";
project my_prog is
...
end my_prog;
```

La recherche des fichiers projets est donné par la variable d'environnement GPR_PROJECT_PATH, par exemple :

```
$ export GPR_PROJECT_PATH=/usr/local/share/gpr
```

La clause importation peut aussi comporter un chemin d'accès direct absolu ou relatif.
Un exemple de fichier projet d'une bibliothèque utilisateur est donné sur github.com/Blady-Com/GPR_Template avec son utilisation.

6. Analyse de la complexité

Pour peu que votre programme soit correct du point de vue de la syntaxe et de la sémantique Ada, gnatmetric va en analyser la complexité.

```
$ gnatmetric figures.adb
```

Le fichier résultat créé par défaut a pour racine le nom du programme augmenté de ".metrix" : figures.adb.metrix.
(voir source figures.adb en page "à savoir")

Il contient des éléments statistiques généraux comme le nombre total de lignes, le nombre de ligne de code, le nombre de lignes de commentaire, le nombres de lignes finissant par un commentaire et le nombre de lignes vides.

```
=== Code line metrics ===
  all lines          : 111
  code lines         : 90
  comment lines      : 13
  end-of-line comments: 4
  blank lines        : 8
```

Puis pour chacune des entités Ada contenues dans le programme, il contient ces mêmes statistiques générales avec en plus des détails sur les instructions et sur la complexité.

```
Figures (procedure body - library item at lines  2: 110)

=== Code line metrics ===
   all lines          : 109
   code lines         : 88
   comment lines      : 13
   end-of-line comments: 4
   blank lines        : 8
```

Les éléments statistiques sur les instructions contiennent le nombre de définitions de types, le nombre de déclarations de sous-programmes, le nombre de corps de sous-programmes, le nombres d'instructions, le nombre de déclarations, le nombre de niveaux d'imbrication d'unités, le nombre d'imbrications d'instructions, le nombre de déclarations et d'instructions connu aussi sous le vocable anglais de Source Lines Of Code (SLOC).

```
=== Element metrics ===
   all type definitions     : 4
   public subprograms       : 1
   all subprogram bodies    : 9
   all statements           : 19
   all declarations         : 60
```

```
   maximal unit nesting     : 2
   maximal construct nesting: 4
   logical SLOC             : 79
```

Les éléments statistiques sur la complexité contiennent le nombre d'instructions de contrôle induisant la complexité, le nombre d'instruction de débranchement (goto ou exit), la somme des deux précédents connue en anglais comme "cyclomatic complexity", le précédent résultat réduit connu en anglais comme "essential complexity", le nombre maximum d'imbrication de boucles.

```
=== Complexity metrics ===
   statement complexity     : 1
   short-circuit complexity : 0
   cyclomatic complexity    : 1
   essential complexity     : 1
   maximum loop nesting     : 0
```

7. Conversion du code source en HTML

Gnathtml.pl est un petit utilitaire codé en langage Perl. Il convertit un fichier source ADA (et ses fichiers de dépendance) en HTML. Les mots-clés sont en gras et les commentaires sont en couleur. Si les informations des références croisées sont trouvées (une compilation complète doit avoir aboutie), les fichiers comporteront également des liens sur chaque référence, c'est à dire que si vous cliquez par exemple sur un type alors sa déclaration serez affichée. Tout ceci est utile pour mettre son code source en ligne sur Internet ou bien pour naviguer dans le code source à la recherche d'une erreur.

À noter que GPS (voir sur Blady) possède une fonction de génération de documentation HTML plus puissante, accessible aussi en ligne avec la commande gnatdoc.

J'utilise avec les options "-f -d" pour avoir les entités locales et les fichiers de dépendance. Mettre l'option "-h" pour afficher le descriptif de la commande.

```
$ gnathtml.pl -f -d toto.adb
```

Les fichiers HTML sont dans le répertoire "html" et s'affichent avec tout navigateur web :

```
$ open html/index.htm
```

8. Optimisations en utilisant les pragma

De façon générale les pragmas sont des directives ajoutées dans le code Ada directement à l'adresse du compilateur pour en modifier son comportement.

Une première famille de pragmas nous permet d'ajouter du code optionnel pour la phase de mise au point (Assert, Debug, Suppress et Unsuppress).

Une seconde famille de pragmas concerne l'affichage d'avertissements (Warning, Unreferenced, No_Return et Obsolescent).

Une dernière famille de pragmas optimise le code ou permet l'utilisation d'un code déjà compilé (Inline, Import, Convention et Export).

a) Le pragma Assert

Il n'y a pas de recette miracle pour garantir l'exactitude d'un programme. Cependant le pragma Assert (défini à partir d'Ada 2005) y contribue en exprimant la conformité de l'implémentation à sa spécification à l'aide d'une simple expresion booléenne.

La syntaxe générale est :

```
pragma Assert([Check =>] boolean_expression[, [Message =>] string_expression]);
```

Exemple :

```
procédure Empiler (X : Element) is
  begin
  -- préconditions
  pragma Assert (not Plein, "La pile est pleine, aucun élément ne peut être ajouté");
  -- empile X ...
  -- postconditions
  pragma Assert (X = Sommet);
  pragma Assert (not Vide);
  end Empiler;
```

Lorsque la condition est fausse l'exception SYSTEM.ASSERTIONS.ASSERT_FAILURE est levée avec le message optionnel.

À noter dans cet exemple, qu'avec Ada 2012 on utilisera plutôt les aspects pré-conditions et post-conditions.

Le pragma Assert peut être aussi bien positionné dans des instructions que dans des déclarations.

Ce fonctionnement peut être modifié avec le pragma Assertion_Policy placé tout au début du code source :

```
pragma Assertion_Policy (CHECK | IGNORE);
```

Avec GNAT, l'option de compilation "-gnata" active les assertions.

b) Le pragma Debug

Le pragma Debug (propre à GNAT) va plus loin en proposant d'exécuter une procédure lorsqu'une condition est réalisée.

La syntaxe générale est :

```
pragma Debug ([CONDITION, ]PROCEDURE_CALL_WITHOUT_SEMICOLON);
```

Exemple :

```
procédure Empiler (X : Element) is
  begin
  -- affiche X si plein
  pragma Debug (Plein, Put_Line (X));
  -- empile X ...
  end Empiler;
```

Ce fonctionnement peut être modifié avec le pragma Debug_Policy :

```
pragma Debug_Policy (CHECK | IGNORE);
```

Avec GNAT, l'option de compilation "-gnata" active les pragma Debug.

c) Les pragmas Suppress et Unsuppress

Nous avons vu au §4 l'utilité d'activer tous les contrôles à l'exécution des dépassements et des allocations (-gnatVa -gnato). Par contre, dans certains cas bien spécifiques, il est nécessaire de désactiver ou d'activer temporairement un contrôle. Les pragmas Suppress et Unsuppress nous permettent respectivement de désactiver ou d'activer un contrôle.

Les contrôles Access_Check, Discriminant_Check, Division_Check, Index_Check, Length_Check, Overflow_Check, Range_Check , Tag_Check provoquent l'émission de l'exception Constraint_Error.

Les contrôles Accessibility_Check, Allocation_Check, Elaboration_Check provoquent l'émission de l'exception Program_Error.

Le contrôle Storage_Check provoque l'émission de l'exception Storage_Error.

All_Checks correspond à l'ensemble des contrôles.

La syntaxe générale est :

```
pragma Suppress(identifier);
pragma Unsuppress(identifier);
```

étendue dans GNAT avec l'option précisant l'entité cible :

```
pragma Suppress (Identifier [, [On =>] Name]);
pragma Unsuppress (Identifier [, [On =>] Name]);
```

étendue également en précisant l'option en ligne, tous les contrôles, la réactivation ou le désactivation temporaire des contrôles :

```
pragma Validity_Checks (string_option | ALL_CHECKS | On | Off);
a       turn on all validity checking options
Cc      turn off/on checking for copies
Dd      turn off/on default (RM) checking
Ee      turn off/on checking for elementary components
Ff      turn off/on checking for floating-point
Ii      turn off/on checking for in params
Mm      turn off/on checking for in out params
Oo      turn off/on checking for operators/attributes
Pp      turn off/on checking for parameters
Rr      turn off/on checking for returns
Ss      turn off/on checking for subscripts
Tt      turn off/on checking for tests
n       turn off all validity checks (including RM)
```

Exemple :

```
  pragma Validity_Checks ("im");
  function "+" (Left : Time; Right : Duration) return Time is
    pragma Unsuppress (Overflow_Check);
  begin
```

```
      return (Left + Time (Right));
    exception
      when Constraint_Error =>
        raise Time_Error;
    end "+";
    type Timed_Delay_Call  is access
      procedure (Time : Duration; Mode : Integer);
    pragma Suppress (Access_Check, Timed_Delay_Call);
```

d) Les pragmas Warning, Unreferenced et No_Return

Nous avons vu au §4 l'utilité d'activer l'ensemble des avertissements à la compilation avec -gnatwa (options ci-dessous marquées d'un +) ou -gnatw.e (toutes les options). Par contre, dans certains cas bien spécifiques, il est nécessaire de désactiver ou d'activer spécifiquement un avertissement.

Le pragma Warning nous permet de désactiver (lettre majuscule) ou d'activer (lettre minuscule) un avertissement parmi la liste :

(l'option par défaut est en caractère gras, l'année d'apparition est entre parenthèses)

Aa	turn off/on all optional info/warnings marked below with +
.A.a+	turn off/**on** warnings for failing assertion (2008)
_A_a+	turn off/**on** warnings for anonymous allocators (2019)
Bb+	turn **off**/on warnings for bad fixed value (not multiple of small)
.B.b+	turn off/**on** warnings for biased representation (2009)
Cc+	turn **off**/on warnings for constant conditional
.C.c+	turn **off**/on warnings for unrepped components (2007)
_C_c	turn off/**on** warnings for unknown Compile_Time_Warning (2020)
Dd	turn **off**/on warnings for implicit dereference
.D.d	turn **off**/on tagging of warnings with -gnatw switch (2013)
e	treat all warnings (but not info) as errors
.e	turn on every optional warning (no exceptions) (2008)
E	treat all run-time warnings as errors (2017)
Ff+	turn **off**/on warnings for unreferenced formal
.F.f	turn **off**/on warnings for suspicious Subp'Access (2015)
Gg+	turn off/**on** warnings for unrecognized pragma (2005)
.g	turn on GNAT warnings (2014, equivalent to -gnatwAao.sI.C.V.X)
Hh	turn **off**/on warnings for hiding declarations
.H.h	turn **off**/on warnings for holes in records (2010)
Ii+	turn off/**on** warnings for implementation unit
.I.i+	turn off/**on** warnings for overlapping actuals (2010)
Jj+	turn **off**/on warnings for obsolescent (annex J) feature (2005)
.J.j+	turn **off**/on warnings for late dispatching primitives (2017)
Kk+	turn **off**/on warnings on constant variable (2005)
.K.k	turn **off**/on warnings for standard redefinition (2013)
Ll	turn **off**/on warnings for elaboration problems
.L.l	turn **off**/on info messages for inherited aspects (2011)
Mm+	turn **off**/on warnings for variable assigned but not read (2005)
.M.m+	turn off/**on** warnings for suspicious usage of modular type (2009)
n	normal warning mode (cancels -gnatws/-gnatwe) (2005)
.N.n	turn **off**/on info messages for atomic synchronization (2012)

Oo turn off/**on** warnings for address clause overlay
.O.o turn **off**/on warnings for out parameters assigned but not read (2008)
Pp+ turn **off**/on warnings for ineffective pragma Inline in frontend
.P.p+ turn **off**/on warnings for suspicious parameter order (2008)
Qq+ turn off/**on** warnings for questionable missing parenthesis (2007)
.Q.q+ turn **off**/on warnings for questionable layout of record types (2018)
Rr+ turn **off**/on warnings for redundant construct
.R.r+ turn **off**/on warnings for object renaming function (2007)
_R_r turn off/on warnings for components out of order (2020)
s suppress all info/warnings
.S.s turn **off**/on warnings for overridden size clause (2010)
Tt turn **off**/on warnings for tracking deleted code (2006)
.T.t turn off/**on** warnings for suspicious contract
Uu+ turn **off**/on warnings for unused entity
.U.u turn **off**/on warnings for unordered enumeration (2010)
Vv+ turn off/**on** warnings for unassigned variable (2005)
.V.v+ turn off/**on** info messages for reverse bit order (2010)
Ww+ turn off/**on** warnings for wrong low bound assumption (2006)
.W.w turn **off**/on warnings on pragma Warnings Off (2008)
Xx+ turn off/**on** warnings for export/import (2005)
.X.x+ turn **off**/on warnings for non-local exception (2007)
Yy+ turn off/**on** warnings for Ada compatibility issues (2006)
.Y.y turn **off**/on info messages for why pkg body needed (2014)
Zz+ turn off/**on** warnings for suspicious unchecked conversion (2005)
.Z.z+ turn off/**on** warnings for record size not multiple of alignment (2014)

Par défaut, les avertissements activés sont :
.a_aB.bC.C_cDFgHi.IJ.JKL.LM.mno.OP.Pq.QR.R.ST.TUvw.Wx.Xyz

La syntaxe générale pour réactiver ou désactiver les avertissements présents est :

pragma Warnings (On | Off);

étendue avec l'option précisant l'entité cible :

pragma Warnings (On | Off, local_NAME);

ainsi que la correspondance directe avec l'option en ligne :

pragma Warnings (static_string_EXPRESSION);

exemple :
pragma Warnings ("h");

étendue également en précisant l'avertissement tel que affiché lors de son émission (joker * permis) :

pragma Warnings (On | Off, static_string_EXPRESSION);

exemple :

```
pragma  Warnings (Off, "*never read*");
```

Le pragma Unreferenced nous permet de désactiver l'avertissement d'une entité déclarée mais non utilisée. Attention, si l'entité est vraiment utilisée un avertissement est tout de même généré.

La syntaxe générale est :

```
pragma Unreferenced (local_NAME {, local_NAME});
pragma Unreferenced (library_unit_NAME {, library_unit_NAME});
```

Le pragma No_Return nous permet de désactiver l'avertissement d'une fonction n'exécutant pas d'instruction "return". Une procédure peut aussi faire l'objet de ce pragma pour avertir qu'elle ne se terminera pas naturellement. Attention, si la fonction ou la procedure se termine tout de même naturellement, une exception Program_Error est levée.

La syntaxe générale est :

```
pragma No_Return(procedure_local_name{, procedure_local_name});
```

Exemples :

```
function AFCB_Allocate (Control_Block : Stream_AFCB) return FCB.AFCB_Ptr is
   pragma Warnings (Off, Control_Block);
begin
   return new Stream_AFCB;
end AFCB_Allocate;

function First_Index (Container : Vector) return Index_Type is
   pragma Unreferenced (Container);
begin
   return Index_Type'First;
end First_Index;

procedure Prog_Exit (Status : Integer);
pragma No_Return (Prog_Exit);
-- Inform compiler and reader that procedure never returns normally
```

e) Le pragma Obsolescent

En programmation, les versions défilent toujours très vite avec ses lots de corrections et d'améliorations. Certaines unités de code sont remplacées par d'autres plus performantes. Ils est toujours ennuyeux de supprimer les anciennes unités de code pour des raisons de compatibilité avec l'existant. Il est aussi ennuyeux de les laisser car de version en version l'accumulation nuit à la maintenabilité. Un compromis est de les laisser temporairement pour les supprimer plus tard lorsque leur utilisation aura disparue.

Le pragma Obsolescent permet d'avertir un utilisateur qu'une unité de code est encore disponible mais va disparaître dans une prochaine version. Un texte peut être ajouté pour guider l'utilisateur sur un code de remplacement.

La syntaxe générale est :

```
pragma Obsolescent  (Entity => NAME [, static_string_EXPRESSION [,Ada_05]]);
```

L'entité nommée est soit une unité de compilation (paquetage, procédure, fonction), soit une variable ou un composant d'article, soit un type ou un énuméré.

À l'endroit de l'utilisation de l'entité, un avertissement est généré avec un message spécifique si celui-ci est présent dans la déclaration du pragma. Le troisième paramètre Ada_05 permet de ne générer d'avertissement qu'avec une compilation en Ada 2005.

Avec GNAT, l'option de compilation "-gnatwj" active l'avertissement d'obsolescence ou "-gnatwa" pour tous les avertissements.

De plus, une erreur est générée si le pragma suivant est déclaré :
"pragma Restrictions (No_Obsolescent_Features);".

Exemples :

```
procedure Initialize;
procedure Initialize (Process_Blocking_IO : Boolean);
pragma Obsolescent
  (Entity => Initialize,
   "passing a parameter to Initialize is not supported anymore");
function To_String
  (Item       : Wide_String;
   Substitute : Character := ' ') return String;
pragma Obsolescent
  ( "(Ada 2005) use Ada.Characters.Conversions.To_String", Ada_05);
```

f) Le pragma Inline

Le choix de créer un sous-programme pour une ou quelques lignes de code n'est pas toujours évident. Parfois ce choix s'impose pour juste des raisons de lisibilité, bien qu'il ne s'y trouve qu'une ligne. Mais ce même choix est pénalisant pour l'exécution qui va englober la ligne de code d'instructions d'entrée et de sortie de sous-programme qui vont donc s'exécuter à chaque appel.

Ada nous permet alors de conserver notre choix d'isoler la ligne dans un sous-programme tout en l'insérant telle quelle lors de l'appel grâce au pragma Inline.

La syntaxe générale est :

```
pragma Inline(name {, name});
```

Les entités nommés dans le pragma sont les noms des sous-programmes dont nous souhaitons mettre le code en ligne.

Avec GNAT l'option -gnatn active la mise du code en ligne pour les procédures désignées par le pragma inline. Les options -gnatwp -Winline provoque un avertissement si le compilateur n'a pas réussi la mise en ligne du code.

Le pragma peut être placé de façon quelconque entre la déclaration du sous-programme et le premier appel, notamment dans la partie privée d'un paquetage.

Exemple :

```
...
  --  Shift routines for the U1/U2/U4 unsigned integers
   function Shift_Left  (Value : U1; Amount : Natural) return U1;
   function Shift_Right (Value : U1; Amount : Natural) return U1;
   function Shift_Left  (Value : U2; Amount : Natural) return U2;
   function Shift_Right (Value : U2; Amount : Natural) return U2;
   function Shift_Left  (Value : U4; Amount : Natural) return U4;
   function Shift_Right (Value : U4; Amount : Natural) return U4;
private
   pragma Inline (Shift_Left);
   pragma Inline (Shift_Right);
...
```

Si le pragma est utilisé pour un sous-programme générique tous les appels de toutes les instances sont concernés. De même, le pragma s'applique à toutes les surcharges d'un sous-programme. Si l'application ne concerne qu'une seule forme il nous faut alors renommer le sous-programme de la forme voulue puis faire porter le pragma sur le nouveau nom.

Exemple :

```
...
  --  Shift routines for the U1/U2/U4 unsigned integers
   function Shift_Left  (Value : U1; Amount : Natural) return U1;
   function Shift_Right (Value : U1; Amount : Natural) return U1;
   function Shift_Left  (Value : U2; Amount : Natural) return U2;
   function Shift_Right (Value : U2; Amount : Natural) return U2;
   function Shift_Left  (Value : U4; Amount : Natural) return U4;
   function Shift_Right (Value : U4; Amount : Natural) return U4;
private
   --  Inline only for the U2 form
   function Shift_Left_U2  (Value : U2; Amount : Natural) return U2 renames Shift_Left;
   function Shift_Right_U2 (Value : U2; Amount : Natural) return U2 renames Shift_Right;
   pragma Inline (Shift_Left_U2);
   pragma Inline (Shift_Right_U2);
...
```

g) Les pragmas Import, Export et Convention

Bien qu'Ada prône la compatibilité au niveau du code source et donc le tout Ada, Ada ne s'isole pas pour autant sur lui même. Le langage est ouvert aux bibliothèques d'autres langages : C, Fortran et même Cobol (voire Java).

Si la liaison avec une bibliothèque étrangère déjà compilée est réalisée simplement en l'ajoutant dans les options de compilation : "-largs -lflorist" par exemple pour la bibliothèque Posix Florist, la liaison avec le code source Ada est réalisée tout aussi simplement à l'aide des pragmas Import et Convention.

Cela permet de réutiliser du code déjà optimisé dans d'autres langages.

La syntaxe générale est :

pragma Import([Convention =>] convention_identifier, [Entity =>] local_name[, [External_Name =>] string_expression]
[, [Link_Name =>] string_expression]);
pragma Convention([Convention =>] convention_identifier,[Entity =>] local_name);

Le paramètre Convention des ces deux pragmas représente le langage d'origine des entités importées C, C_Pass_By_Copy (uniquement pour le pragma Convention), COBOL, Fortran (Ada et Intrinsic peuvent également être employés dans des utilisations plus spécifiques, voir le manuel Ada 2005 §6.3.1 Conformance Rules).
Le paramètre Entity est pour le pragma Import le nom de l'objet importé comme un sous-programme ou une variable, il est pour le pragma Convention le plus souvent un type de données.

Les paramètres optionnels External_Name ou Link_Name définissent une chaîne représentant le nom utilisé par le langage étranger ou par l'éditeur de liens.

Exemple :

```
with Interfaces.C;
...
   type struct_sigaction is record
      sa_handler : System.Address;
      sa_mask    : sigset_t;
      sa_flags   : int;
   end record;
   pragma Convention (C, struct_sigaction);
   type struct_sigaction_ptr is access all struct_sigaction;
...
   function sigaction
     (sig  : Signal;
      act  : struct_sigaction_ptr;
      oact : struct_sigaction_ptr) return int;
   pragma Import (C, sigaction, "sigaction");
```

Il est tout aussi simple d'exporter du code Ada vers un langage étranger.

La syntaxe générale est :

pragma Export([Convention =>] convention_identifier, [Entity =>] local_name[, [External_Name =>] string_expression]
[, [Link_Name =>] string_expression]);

Exemple :

```
   Main_Priority : Integer := -1;
   pragma Export (C, Main_Priority, "__gl_main_priority");
```

Si le pragma est utilisé pour un sous-programme générique tous les appels de toutes les instances sont concernés. De même, le pragma s'applique à toutes les surcharges d'un sous-programme. Si l'application ne concerne qu'une seule forme il nous faut alors renommer le sous-

programme de la forme voulue puis faire porter le pragma sur le nouveau nom (voir exemple du pragma inline).

h) Le fichier gnat.adc

Lorsqu'un pragma s'applique à l'ensemble de la compilation comme "pragma Restrictions (No_Obsolescent_Features);", il est possible de le mettre une seule fois dans le fichier gnat.adc (placé dans le répertoire courant de la compilation ou dans un emplacement spécifique "path" indiqué par l'option -gnatecpath) alors il s'appliquera à l'ensemble des codes sources compilés, sauf si l'option -gnatA est positionnée.

9. Les options de gnatmake

Les options suivantes vont permettre de contrôler les fichiers à compiler :
(l'année d'apparition est entre parenthèses)

--version Display version and exit (2012)
Affiche la version.
--help Display usage and exit (2012)
Affiche l'aide.
-c Compile only
Pas de bind ni de link.
-d Display compilation progress (2014)
Affiche le progression de la compilation
-D dir Specify dir as the object directory (2005)
Met les .o et .ali dans "dir" (doit être déjà créé) sinon ils vont dans le répertoire courant.
-f Force recompilations of non predefined units
Recompile même les codes sources non modifiés.
-k Keep going after compilation errors
Continue la compilation des codes sources même en cas d'erreurs.
-m Minimal recompilation
Ne recompile pas les codes sources incluants seulement des modifications de blancs (espaces), lignes vides, tabulations ou commentaires.
-M List object file dependences for Makefile
Donne les dépendances pour chaque code source. Ne fonctionne que si la compilation est à jour.
-n Check objects up to date, output next file to compile if not
Affiche le fichier à compiler sinon rien.
-o name Choose an alternate executable name
Prend "name" comme nom de fichier pour le programme exécutable.
-p Create missing obj, lib and exec dirs (2007)
Crée les répertoires obj, lib et exec si manquants dans le projet utilisé avec -P.
-P proj Use GNAT Project File proj
Utilise les options du projet "proj.gpr".
-s Recompile if compiler switches have changed
Recompile les codes sources non modifiés s'ils n'ont pas été compilés avec les mêmes options.
-u Unique compilation. Only compile the given file.
Ne compile que le fichier présent dans la commande.
-v Display reasons for all (re)compilations
Affiche les raisons de la compilation de chaque code source.
-ws Suppress all warnings (2007)
Supprime les avertissements.
-z No main subprogram (zero main)
Compile une bibliothèque sans programme principal.
--keep-temp-files Keep temporary files (2016)
Ne supprime pas les fichiers temporaires "config pragma", "mapping" ou "project path".
-Idir Like -aIdir Specify source files search path -aOdir Specify library/object files search path
Indique le chemin vers une bibliothèque Ada.
-I- Don't look for sources & library files in the default directory
Ne recherche pas les sources et bibliothèques dans le répertoire courant.

-Ldir Look for program libraries also in dir
Indique le chemin vers une bibliothèque Ada.
-gnat83 Ada 83 mode
-**gnat95** Ada 95 mode
-gnat2005 Ada 2005 mode
-gnat2012 Ada 2012 mode (default) (2010)
-gnat2022 Ada 2022 mode (2021)
Défini la syntaxe du code source du standard Ada (mode Ada 2012 par défaut).

10.L'utilitaire de formatage du code source : gnatpp

Les formateurs de code source, en anglais "beautifier" ou "pretty printer", existent sans être vraiment populaires. Pourtant, leur utilisation systématique permettra d'améliorer la lisibilité et maintenabilité du code. Ainsi le suivi des modifications est simplifié car les différences apparaissant ne sont réellement que des différences de codage et ne sont pas mélangées avec des différences de présentations.

L'utilitaire gnatpp produit par défaut la syntaxe des sources de GNAT. Les valeurs par défaut sont :
- indentation avec retrait de 3 caractères
- longueur maximale des lignes fixée à 79 caractères
- mots clés du langage en minuscule
- attributs et pragmas avec premier caractère en majuscule
- casse des identificateurs telle qu'ils apparaissent pour la première fois

J'ajoute les options :
-M99 : pour allonger la taille maximale des lignes à 99 caractères
-rnb : remplace le fichier source par le nouveau formaté sans sauvegarde
-W8 : encode le nouveau source formaté en UTF-8
-cargs -gnatW8 : interprète le format UTF-8

```
$ gnatpp -M99 -rnb -W8 hello4.adb -cargs -gnatW8
```

Le reformage du code Ada est désactivé avec la séquence :
--!pp off
Il est réactivé avec la séquence :
--!pp on

Ces séquences peuvent être remplacées par des séquences particulières passées en paramètre de gnatpp comme "--pp-off=xxx" en utilisant "--xxx" pour désactiver le reformage et "--pp-on=yyy" en utilisant "--yyy" pour réactiver le reformage.

11.L'utilitaire de nettoyage des fichiers issus de la compilation : gnatclean

La compilation avec gnat produit toutes sortes de fichiers nécessaires à la construction du code exécutable, il est parfois nécessaire de faire du nettoyage pour repartir juste des codes sources. L'utilitaire gnatclean supprime les fichiers produits par le compilateur à partir d'un fichier source (sans son extension) :

```
$ gnatclean essai14
"./essai14.ali" has been deleted
"./essai14.o" has been deleted
"essai14" has been deleted
"b~essai14.ads" has been deleted
"b~essai14.adb" has been deleted
"b~essai14.ali" has been deleted
"b~essai14.o" has been deleted
```

Ou à partir d'un projet GPR :

```
$ gnatclean -P test04.gpr -XLIBRARY_TYPE=static
"gtkada/obj/test04.ali" has been deleted
"gtkada/obj/test04.o" has been deleted
"gtkada/obj/test04" has been deleted
```

L'option "-n" permet de simuler les suppressions :

```
$ gnatclean -n essai14
./essai14.ali
./essai14.o
essai14
b~essai14.ads
b~essai14.adb
b~essai14.ali
b~essai14.o
```

Pascal Pignard, février-décembre 2008, juillet 2009, septembre-décembre 2010, mars-octobre 2012, juin 2013, août 2014, décembre 2014, juillet 2015, juin 2016, juin 2017, juin 2018, mai-août 2019, mai-juillet 2020, juillet 2021.