

Apprenez à programmer avec Ada

Par Kaji9



www.openclassrooms.com

*Licence Creative Commons 6 2.0
Dernière mise à jour le 7/10/2013*

Sommaire

Sommaire	2
Lire aussi	9
Apprenez à programmer avec Ada	11
Partie 1 : Premiers pas avec Ada	12
Programmation, algorithmique et Ada ?	12
Qu'est-ce qu'un programme ?	12
Qu'appelle-t-on programme ?	12
Alors qu'allons-nous faire ?	13
Comment réaliser un programme ?	13
Pourquoi Ada et pas autre chose ?	13
Algorithmique	13
Pourquoi Ada ?	14
En résumé :	14
Les logiciels nécessaires	15
IDE et compilateur : kesako ?	15
Le compilateur	15
L'IDE ou EDI	15
Télécharger et installer Adagide	15
Téléchargement	15
Installation	15
Télécharger et installer GNAT	15
Téléchargement	15
Installation	15
Télécharger et installer GPS	16
En résumé :	16
Notre premier programme en Ada	16
Découvrir son IDE en quelques secondes	17
Soyons rapide avec Adagide	17
Pour les utilisateurs de GPS (plus long)	17
Notre premier programme	18
Un petit copier-coller !	18
Compiler, créer... lancer !	18
Mon dieu, qu'ai-je fait ?	18
Le corps du programme : la procédure Hello	19
Les Packages avec With et Use	19
Une dernière remarque qui a son importance	20
Exercices	20
Exercice 1	20
Exercice 2	20
Exercice 3	20
En résumé :	21
Partie 2 : Ada, notions essentielles	22
Variables I : Typage et affectation	22
Déclaration de variables	22
Différents types	22
Les types Integer et Natural	22
Le type Float	23
Le type Character	24
Affectation	24
Affectation par le programmeur (ou le programme)	25
Affectation par l'utilisateur	25
L'instruction Skip_line	25
Compléments	25
Constantes	26
Attributs	26
Bloc de déclaration	26
En résumé :	27
Variables II : Opérations	28
Opérations sur les Integer et les Natural	28
Opérations sur les float	28
Opérations élémentaires	28
Ne faites pas d'erreur de casting	28
Opérations mathématiques	28
Opérations sur les character	29
Exercices	29
Exercice 1	29
Exercice 2	29
Exercice 3	30
Exercice 4	30
En résumé :	30
Les conditions I	30
Conditions simples avec IF	31
Un début en douceur	31
Une première alternative	31

Conditions multiples avec IF	31
Conditions multiples avec CASE	32
Tester de nombreux cas	32
Ne rien faire	32
Les opérateurs de comparaison et d'appartenance	32
Les opérateurs de comparaison	32
L'opérateur d'appartenance	33
Pour les utilisateurs de la norme Ada2012	33
Les expressions IF	33
Les expressions CASE	34
Toujours plus complexe	34
En résumé :	34
Les conditions II : les booléens	34
Introduction aux booléens	35
Un bref historique	35
Qu'est-ce qu'un booléen ?	35
Les opérateurs booléen	35
La négation avec Not	35
Les opérations Or et And	36
L'instruction OR	36
L'instruction AND	36
L'opération XOR (optionnel)	36
Exercice	36
Priorités booléennes et ordre de test (Supplément)	37
Priorités avec les booléens	37
Ordre de test	37
En résumé :	38
Les boucles	39
La boucle Loop simple	39
Principe général	39
Arrêter une itération	39
Nommer une boucle	40
La boucle While	40
La boucle For	40
Les antiquités : l'instruction goto	41
Boucles imbriquées	41
Méthode avec une seule boucle (plutôt mathématique)	41
Méthode avec deux boucles (plus naturelle)	41
Exercices	42
Exercice 1	42
Exercice 2	42
Exercice 3	42
Exercice 4	42
Exercice 5	42
Exercice 6	42
En résumé :	42
Procédures et fonctions I	43
Les procédures	43
Procédure sans paramètre	43
Procédure avec un paramètre (ou argument)	44
Procédure avec plusieurs paramètres (ou arguments)	44
Les fonctions	45
Une bête fonction	45
Une fonction un peu moins bête (optionnel)	46
Bilan	47
Prédefinir ses paramètres	47
In, Out, In Out	48
Paramètres de procédure	48
Paramètres de fonction	49
En résumé :	49
[TP] Le craps	49
Les règles du craps	50
Cahier des charges	50
Simuler le hasard (ou presque)	50
Un plan de bataille	51
Une solution	51
Pistes d'amélioration :	52
Partie 3 : Ada, les types composites	53
Les tableaux	53
Les types composites, c'est quoi ?	53
Tableaux unidimensionnels	53
Problème	53
Création d'un tableau en Ada	54
Attributs pour les tableaux	55
Tableaux multidimensionnels	56
Tableaux bidimensionnels	56
Tableaux tridimensionnels et plus	56
Et mes attributs ?	57
Et mes agrégats ?	57
Des tableaux un peu moins contraints	57
Un type non-contraint ou presque	57
Affectation par tranche	57

Déclarer un tableau en cours de programme	58
Quelques exercices	58
Exercice 1	58
Exercice 2	59
Exercice 3	59
Pour les utilisateurs de la norme Ada2012	60
Boucle FOR OF	60
Expressions quantifiées	61
En résumé :	61
Les chaînes de caractères	61
Présentation des Chaînes de Caractères	62
Déclaration et affectation d'un string	62
Quelques opérations sur les strings	62
Accès à une valeur	62
Accès à plusieurs valeurs	62
Modifier la casse	63
Concaténation	63
Transformer une variable en string et inversement	63
Comparaison	63
Saisie au clavier	64
Chaînes de caractères non contraintes	64
Déclarer des strings illimités !	64
Opérations sur les unbounded_string	64
En résumé :	65
La programmation modulaire I : les packages	66
Les fichiers nécessaires	66
Notre première procédure... empaquetée	66
Variables et constantes globales	68
Trouver et classer les fichiers	68
Les packages fournis avec GNAT	68
Organiser nos packages	68
Petite astuce pour limiter les fichiers générés	69
Compléter notre package (exercices)	69
Cahier des charges	69
Solutions	70
Vecteurs et calcul vectoriel (optionnel)	72
Qu'est-ce exactement qu'un T_Vecteur ?	72
Calcul vectoriel	72
En résumé :	73
Les fichiers	73
Ouvrir / Fermer un fichier texte	74
Package nécessaire	74
Le type de l'objet	74
Fermer un fichier	74
Ouvrir un fichier	74
Le paramètre Mode	75
Lecture seule	75
Écriture seule	75
Ajout	75
Opérations sur les fichiers textes	75
Mode lecture seule : In_File	75
Mode écriture : Out_File / Append_File	77
Autres opérations	77
Les fichiers binaires séquentiels	77
Les fichiers binaires directs	78
Les répertoires	79
Chemins absous et relatifs	79
Indiquer le chemin d'accès	79
Gérer fichiers et répertoires	79
Quelques exercices	80
Exercice 1	80
Exercice 2	80
Exercice 3	80
En résumé :	81
Créer vos propres types	82
Créer à partir de types prédéfinis	82
Sous-type comme intervalle	82
Types modulaires	82
Énumérer les valeurs d'un type	82
Les types structurés	83
Déclarer un type «construit»	83
Ordre des déclarations	84
Déclarer et modifier un objet de type structuré	84
22 ! Rev'là les tableaux !	85
Les types structurés : polymorphes et mutants !	85
Les types structurés polymorphes	85
Les types structurés mutants	86
En résumé :	88
[TP] Logiciel de gestion de bibliothèque	89
Cahier des charges	89
Quelles données pour quels types de données ?	89
Quelle architecture pour les fichiers	89

Quelles fonctionnalités pour quelles fonctions et procédures ?	89
Architecture du code source	89
Conception du programme (suivez le guide)	90
Création des types	90
Affichage d'une œuvre	90
Saisie d'une œuvre	90
Gestion des fichiers	90
Les commandes	91
Solutions possibles	91
Pistes d'amélioration :	95
Les pointeurs I : allocation dynamique	96
Mémoire, variable et pointeur	96
Le type access	97
Déclarer un pointeur	97
Que contient mon pointeur ?	97
Comment accéder aux données ?	97
Opérations sur les pointeurs	98
Une erreur à éviter	98
Libération de la mémoire	98
Un programme (un peu) gourmand	98
Un problème de mémoire	99
Résolution du problème	100
Exercices	101
Exercice 1	101
Exercice 2	101
En résumé :	101
Les pointeurs II	101
Cas général	102
Pointeurs généralisés : pointer sur une variable	102
Pointeur sur une constante et pointeur constant	102
Pointeur sur pointeur	103
Pointeur comme paramètre	103
Pointeur sur un programme (optionnel)	104
Un exemple simple	104
Un exemple de la vie courante	104
Un exemple très... mathématique	105
Exercices	106
Exercice 1	106
Exercice 2	106
Exercice 3	107
Exercice 4	107
Exercice 5 (Niveau Scientifique)	107
En résumé :	108
Fonctions et procédures II : la récursivité	108
Une première définition	109
Exemple d'algorithme récursif	109
Notre première fonction récursive	110
Énoncé	110
Indications	110
Une solution possible	110
Algorithme de recherche par dichotomie	111
Principe	111
Mise en œuvre	111
Solution	111
Quelques exercices	112
Exercice 1	112
Exercice 2	112
Exercice 3	112
Exercice 4	113
En résumé :	113
Les Types Abstraits de Données : listes, files, piles...	113
Qu'est-ce qu'un Type Abstrait de Données ?	114
Un premier cas	114
Autres cas	114
Primitives	115
Les piles	115
Création du type T_Pile	115
Création des primitives	116
Jouons avec le package P_Pile	117
Les files	118
Implémentation	118
Amusons-nous encore	119
Les listes chaînées	119
Quelques rappels	119
Le package Ada.Containers.Doubly_Linked_Lists	120
Le package Ada.Containers.Vectors	121
En résumé :	122
[TP] Le jeu du serpent	124
Cahier des charges	124
Fonctionnalités	124
Organisation des types et variables	124
Un package bien utile	124

Le package NT_Console	124
Le contenu en détail	129
... et encore un autre !	130
Quelques indications	130
Jouer en temps réel	130
Comment afficher un serpent et une zone de jeu en couleur ?	131
Par où commencer ?	131
Une solution possible	131
Pistes d'amélioration :	135
Partie 4 : Ada : Notions avancées et Programmation Orientée Objet	135
Algorithmique : tri et complexité	136
Algorithmes de tri lents	136
Tri par sélection (ou par extraction)	136
Tri par insertion	137
Tri à bulles (ou par propagation)	137
Algorithmes de tri plus rapides	138
Tri rapide (ou Quick Sort)	138
Tri fusion (ou Merge Sort)	138
Tri par tas	139
Théorie : complexité d'un algorithme	142
Complexité	142
L'écriture O	142
Quelques fonctions mathématiques	142
Mesures de complexité des algorithmes	143
Un algorithme pour mesurer la complexité... des algorithmes	143
Traiter nos résultats	144
En résumé :	145
Variables III : Gestion bas niveau des données	146
Représentation des nombres entiers	146
Le code binaire	146
Conversions entre décimal et binaire	146
Retour sur le langage Ada	147
Représentation du texte	148
Représentation des nombres décimaux en virgule flottante	149
Représentation des float	149
Implications sur le langage Ada	150
En résumé :	151
La programmation modulaire II : Encapsulation	152
Qu'est-ce qu'un objet ?	152
Une première approche	152
Posons le vocabulaire	152
De nouvelles contraintes	153
Un package... privé	153
Partie publique / partie privée	153
Visibilité	154
Un package privé et limité	155
Que faire avec un type PRIVATE ?	155
Restreindre encore notre type	155
Exercices	156
Exercice 1	156
Exercice 2	156
En résumé :	156
La programmation modulaire III : Généricité	156
Généricité : les grandes lignes	157
Que veut-on faire ?	157
Plan de bataille	157
Un dernier point de vocabulaire	157
Créer et utiliser une méthode générique	157
Créer une méthode générique	157
Utiliser une méthode générique	158
Paramètres génériques de types simples et privés	159
Types génériques simples	159
Types génériques privés	159
Paramètres génériques de types composites et programmes	159
Tableaux génériques	159
Pointeurs génériques	160
Paramètre de type programme : le cas d'un paramètre LIMITED PRIVATE	160
Packages génériques	161
Exercice	161
Solution	162
Application	162
En résumé :	163
La programmation modulaire IV : Héritage et dérivation	163
Pour bien commencer	164
Héritage : une première approche théorique	164
Héritage : une approche par l'exemple	164
Héritage	165
Héritage de package simple	165
Héritage avec des packages privés	166
Héritage avec des packages génériques	167
Dérivation et types étiquetés	168
Créer un type étiqueté	168

Et nos méthodes ?	168
Un autre avantage des classes	170
En résumé :	170
La programmation modulaire V : Polymorphisme, abstraction et héritage multiple	170
Polymorphisme	171
Méthodes polymorphes	171
Objets polymorphes	171
Abstraction	173
Types abstraits	173
Méthodes abstraites	173
Héritage multiple	174
Comment Ada gère-t-il l'héritage multiple ?	174
Réaliser des interfaces Ada	175
En résumé :	177
La programmation modulaire VI : Finalisation et types contrôlés	177
Objectifs et prérequis	178
De quoi parle-t-on exactement ?	178
Comment s'y prendre ?	178
Mise en œuvre	178
Mise en place de nos types	179
Mise en place de nos méthodes	179
En résumé :	180
[TP] Bataille navale	181
Règles du jeu	181
Le déroulement	181
Les navires	181
Les statistiques	181
Cahier des charges	181
Gameplay	181
Les calculs	181
POO	182
Une solution possible	182
L'organisation	182
Le code	182
Pistes d'amélioration :	194
Les exceptions	194
Fonctionnement d'une exception	195
Vous avez dit exception ?	195
Le fonctionnement par l'exemple	195
Traitement d'une exception	195
Le bloc EXCEPTION	195
Où et quand gérer une exception	196
Exceptions prédéfinies	196
Exceptions standards	196
Autres exceptions prédéfinies	197
Créer et lever ses propres exceptions	197
Déclarer une exception	197
Lever sa propre exception	197
Propagation de l'exception	198
Rectifier son code	198
Assertions et contrats	199
Assertions	199
Programmation par contrats (Ada2012 seulement)	200
En résumé :	200
Multitasking	201
Parallélisme, tâches et types tâches	201
Multitasking ou l'art de faire plusieurs tâches à la fois	201
Les tâches en Ada	201
Le type tâche	203
Communication inter-tâche directe	204
Le parallélisme, comment ça marche ?	204
Les entrées	205
Les synchronisations sélectives	206
Gardes et compléments	206
Communication inter-tâche indirecte	207
Les types protégés	207
Les sémaphores	208
Les moniteurs	209
Compléments : priorités et POO	210
Priorités	210
Quand la programmation orientée objet rejoint la programmation concurrente	211
Programme multitâche et processeur multicœur (Ada 2012 uniquement)	212
Exercices fondamentaux	212
Modèle des producteurs et consommateurs	212
Modèle des lecteurs et rédacteurs	213
Le dîner des philosophes	215
En résumé :	216
Interfaçage entre Ada et le C	216
Quelques préparatifs	217
Logiciels nécessaires	217
Quelques rudiments de C	217
Packages nécessaires	218

Hello World : du C à l'Ada	218
Notre programme en C	218
Notre programme Ada avec les normes 95 ou 2005	218
Notre programme Ada avec la norme 2012	219
Quelques menus problèmes	219
Procédure avec paramètres	219
Avec un type structuré	220
En résumé :	221
Partie 5 : Ada et GTK : la programmation évènementielle	222
GTKAda : introduction et installation	222
Vous avez dit GTK ?	222
Qu'est-ce que GTK ?	222
GTK, GTKAda, GDK, Glib et toute la famille	222
Pourquoi ce choix ?	223
Télécharger et installer GTKAda	223
Télécharger	223
Installer	224
Configurer votre IDE	224
Un premier essai	225
En résumé :	225
Votre première fenêtre	225
Analysons notre code	226
Code GtkAda minimal	226
Créer une fenêtre	226
Personnaliser la fenêtre	227
Changer de type de fenêtre	227
Définir les paramètres avec Set_#	227
Ajout d'un widget	229
Qu'est-ce qu'un widget ?	229
Ajouter un bouton	229
Personnaliser le bouton	230
Ajouter un second bouton ?	230
Retour sur la POO	231
Méthode brutale	231
Méthode subtile	231
En résumé :	232
Les conteneurs I	232
Des conteneurs pour... contenir !	233
Qu'est-ce qu'un conteneur ?	233
Présentation de différents conteneurs	233
Les alignements	234
Fiche d'identité	234
Créer un GTK_Alignment	234
Le padding	235
Les boîtes	236
Boîtes classiques	236
Boîtes à boutons	239
Les tables	240
Fiche d'identité	240
Créer une table de widgets	240
Ajouter des widgets	241
Les paramètres supplémentaires	241
Le widget pour position fixe	242
Fiche d'identité	242
Utilisation des GTK_Fixed	242
En résumé :	242
Les signaux	244
Le principe	244
Les problèmes de la programmation évènementielle	244
Le principe Signal-Procédure de rappel	244
Connecter un signal à un callback	244
Fermer proprement le programme	244
Utiliser le bouton	245
Interagir avec les widgets	246
Un callback à deux paramètres	246
Un nouveau package de callbacks	247
La connexion	247
Perfectionner encore notre code	248
Autres packages de callback	249
GDK et les événements	249
Le clic droit	249
Le double clic	250
Le clavier	251
En résumé :	252
Les widgets I	253
Les étiquettes	253
Fiche d'identité	253
Quelques méthodes des GTK_Label	253
Pour une mise en forme plus poussée	254
Les images	255
Fiche d'identité	255
Méthodes	255

Exercice : créer un bouton avec icône	256
Les zones de saisie	257
La saisie de texte sur une ligne : les entrées	257
Saisie de texte multiligne	258
Saisie numérique	263
Saisie numérique par curseur	263
D'autres boutons	264
Boutons à bascule	264
Boutons-liens	264
Boutons à cocher	265
Boutons radios	265
Widgets divers	265
Les séparateurs	266
Les flèches	266
Le calendrier	266
Les barres de progression	267
En résumé :	267
Les Widgets II : les boîtes de dialogue	267
Message	268
Fiche d'identité	268
Méthodes	268
Les signaux	268
À propos	269
Fiche d'identité	269
Méthodes	269
Sélection de fichier	270
Fiche d'identité	270
Méthodes des GTK_File_Chooser_Dialog	270
Les GTK_File_Chooser (sans le Dialog)	271
Deux widgets supplémentaires	272
Signaux des Gtk_File_Chooser	272
Sélection de police	272
Fiche d'identité	272
Méthodes	272
Sélection de couleur	273
Fiche d'identité	273
Méthodes	273
Signaux	273
Cas général	273
Fiche d'identité	273
Méthodes	273
Signaux	273
En résumé :	274
[TP] Le démineur	275
Règles du jeu et cahier des charges	275
Quelques rappels	275
Règles retenues	275
Quelques ressources	275
Un petit coup de main	275
Premier objectif : détruire une case	275
Second objectif : placer un drapeau	276
Troisième objectif : passer du bouton unique à la grille	276
Quatrième objectif : destruction de cases en cascade	276
Une solution possible	276
Les spécifications	276
Le corps des packages	278
La procédure principale	281
Pistes d'amélioration :	281
Les conteneurs II	281
Les onglets	282
Fiche d'identité	282
Méthodes	282
Les barres de défilement	284
Fiche d'identité	284
Exemples d'utilisation	284
Méthodes	286
Les panneaux	286
Fiche d'identité	286
Méthodes	287
Les cadres	287
Les cadres simples	287
Les cadres d'aspect	288
Les extenseurs	288
Fiche d'identité	288
Méthodes	288
Les boîtes détachables	289
Fiche d'identité	289
Méthodes	289
En résumé :	289
Les widgets III : barres et menus	289
La barre de menu	290
Créer une barre de menu	290
Créer et ajouter des items	290

Créer et ajouter un menu déroulant	291
Améliorer la barre de menu	292
Créer un menu en image	292
Créer un sous-menu	292
Proposer un item à cocher dans le menu	293
Organiser vos menus	294
La barre d'icônes	294
Fiches d'identité	294
Créer notre barre d'icônes	294
Améliorer les barres d'icônes	295
Combiner menus et icônes	296
La barre de statut	297
Fiche d'identité	297
Méthodes et fonctionnement	297
Exemples d'utilisation	298
Le menu déroulant	299
Fiche d'identité	299
Méthodes	299
Menu déroulant avec saisie	300
En résumé :	300
[TP] Démineur (le retour)	300
Cahier des charges	301
Objectifs	301
Widgets nécessaires	301
Une solution possible	301
Les spécifications	301
Le corps des packages	305
La procédure principale	310
Pistes d'amélioration	310

Apprenez à programmer avec Ada



Apprenez à programmer avec Ada

Par Kajj9

Mise à jour : 07/10/2013

Difficulté : Intermédiaire

Durée d'étude : 2 mois

[cc] Creative Commons

5 visites depuis 7 jours, classé 16/807

Vous voudriez apprendre à programmer mais ne savez pas par où commencer ? Vous avez commencé à apprendre un autre langage mais vous vous embrouillez dans les accolades et autres symboles bizarroïdes ? Ou encore vous souhaitez apprendre un nouveau langage de programmation ? Alors ce tutoriel est fait pour vous.

Le langage Ada est certainement le meilleur langage pour apprendre à programmer : pour peu que vous connaissiez deux ou trois mots d'anglais, il vous sera facile de lire un code en Ada. Moins abstrait que beaucoup de langages, mais toutefois rigoureux, facilement compréhensible et lisible, même par un novice, le langage Ada vous permettra de comprendre les logiques propres à la programmation et vous donnera tout de suite de bonnes habitudes.

Qui plus est, le langage Ada ne présente ni plus ni moins de fonctionnalités que les langages les plus connus (C, C++, Java, Python...). Il est seulement différent et, je me répète, plus accessible aux débutants. Alors si vous êtes intéressé, nous allons pouvoir débuter de tutorial 😊 Celui-ci est organisé en cinq parties :

- La partie I constitue une introduction à la programmation et vous accompagne dans votre première prise en main d'Ada.
- Les parties II, III et IV traitent du langage Ada en lui-même et présenteront une difficulté progressive : notions de base, types composés puis programmation orientée objet.
- La partie V constitue une mise en pratique en vous proposant de créer des programmes fenêtrés avec GTK et Ada.

Le cours sera ponctué de travaux pratiques. Ce sont des chapitres vous proposant un projet (parfois ludique) allant du logiciel de gestion de vos DVD à des jeux de dés ou de démineur. Ils seront pour vous l'occasion de mettre en application vos connaissances acquises.

Partie 1 : Premiers pas avec Ada

Dans cette toute première partie, la plus courte, nous allons poser les bases en répondant à quelques questions existentielles : qu'est-ce que la programmation ? Quelle différence avec l'algorithme ? Qu'est-ce qu'un langage de programmation ? Comment faire pour écrire un programme en Ada ? Nous verrons également comment installer les quelques logiciels nécessaires ainsi que leur fonctionnement. Enfin, nous créerons notre tout premier programme en Ada et tenterons de comprendre comment ceux-ci sont organisés.

Programmation, algorithmique et Ada ?

Nous allons dans ce chapitre répondre à quelques questions préliminaires. Qu'est-ce qu'un programme ? À quoi cela ressemble-t-il ? Comment crée-t-on un programme ? Qu'est-ce que l'algorithme ? Pourquoi utiliser Ada et non pas un autre langage ? Ce que j'apprends ici sera-t-il utilisable avec un autre langage ?

Bref, beaucoup de réponses existentielles trouveront (je l'espère) leur réponse ici. Ce chapitre, relativement court et sans difficultés, permettra de poser un certain nombre d'idées et de termes. Pour bien commencer, mieux vaut en effet que nous ayons le même vocabulaire. 😊

À ceux qui auraient déjà des notions de programmation, vous pouvez bien entendu éviter ce chapitre. Je ne saurais trop toutefois vous conseiller d'y jeter un œil, histoire de remettre quelques idées au clair.

Qu'est-ce qu'un programme ?

Avant de commencer, il est important que nous nous entendions sur les termes que nous allons utiliser.

Qu'appelle-t-on programme ?

Les programmes sont des fichiers informatiques particuliers. Sous Windows, ce sont généralement des fichiers se terminant par l'extension .exe ; sous MacOS ce sont globalement des fichiers .app ; sous Linux ce sont schématiquement des fichiers .bin. Cette explication est très (TRES) schématique, car d'autres types de fichiers peuvent correspondre à des programmes.

Ces fichiers ont comme particularité de ne pas seulement contenir des données, mais également des instructions. Lorsque vous ouvrez un fichier .exe sous windows, celui-ci transmet à votre ordinateur une liste de tâches à effectuer comme :

- ouvrir une fenêtre
- afficher du texte
- effectuer des calculs
- ...

Cette liste d'instructions envoyées au processeur, forme alors ce que l'on appelle un processus.

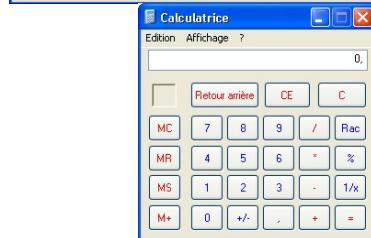


On fait généralement un amalgame entre programme et processus. Le programme est, rappelons-le, un fichier. Le processus correspond aux instructions effectuées par le processeur et qui viennent du programme. Mais, en général, on appelle tout cela programme 😊

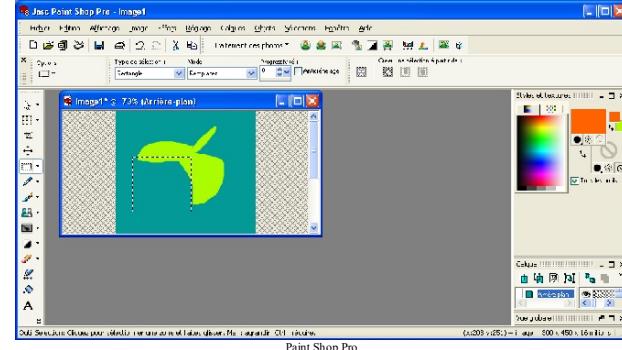
Vous trouverez ci-dessous trois exemples de programmes.



Mozilla Firefox

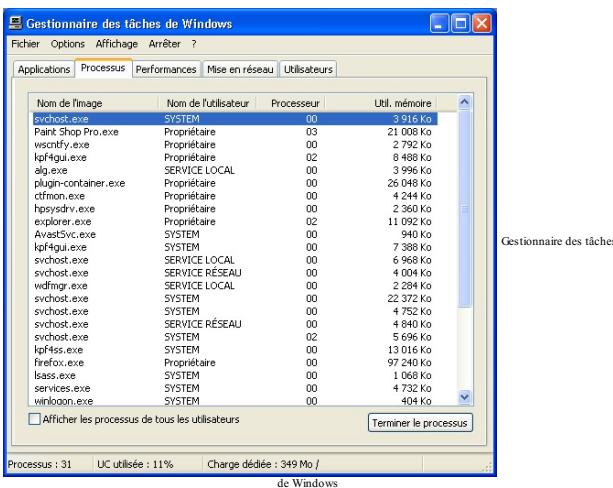


Calculatrice de Windows



Paint Shop Pro

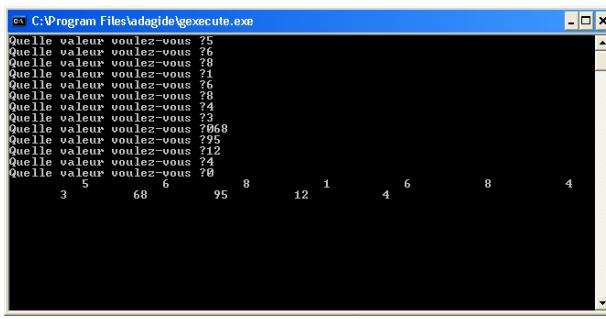
Attention, tous les programmes ne sont pas nécessairement visibles. Beaucoup n'ouvrent même pas de fenêtre ! Pour s'en rendre compte, sous Windows, appuyez simultanément sur les touches Alt + Ctrl + Suppr, puis cliquez sur le second onglet, cela affichera la liste de tous les processus en cours.



Bon nombre d'entre eux effectuent des tâches tranquillement sans que vous ne vous en rendiez compte.

Alors qu'allons-nous faire ?

Eh bien, programmer (en Ada ou avec tout autre langage) c'est tout simplement créer des programmes. Mais je vous aiient toutes de suite, malgré tout le talent de votre humbre guide, vous ne confectionnerez pas un jeu vidéo en 3D d'ici la fin de ce cours. La plupart des logiciels, comme le navigateur internet que vous utilisez actuellement, exigent la participation de de nombreuses personnes et de nombreuses autres compétences ne relevant pas de ce cours (modélisation 3D par exemple). Les premiers programmes que nous allons concevoir seront des programmes en console. C'est-à-dire qu'ils ressembleront à la figure suivante:



Nous ne pourrons utiliser que le clavier ! Pas de souris ou de joystick ! Pas d'images ou de vidéos ! Pas de 3D puisqu'il n'y aura même pas de 2D ! Que du texte blanc sur fond noir.

 Aaargh ! Mais, c'est possible ? Ces horreurs existent encore ?

Bon, c'est vrai qu'aujourd'hui on a tendance àoublier la console, mais il est nécessaire de passer par ce stade pour apprendre les bases de la programmation. Nous pourrons ensuite nous atteler à des programmes plus conséquents (avec des boutons, des images, la possibilité d'utiliser la souris...). Mais il faudra être patient.

Comment réaliser un programme ?

Comment réaliser un programme ?

Alors comment faire pour créer votre propre programme ? Il faut avant tout savoir comment est constitué un programme (ou tout autre type de fichier). Vous le savez peut-être déjà, mais un ordinateur a un langage très très basique. Il ne connaît que deux chiffres : 1 et 0 ! Donc tout fichier (et donc tout programme) ne peut ressembler qu'à ceci :

10001001110101101101101101

Ca rappelle Matrix ! Mais comment je fais moi ? J'y comprends rien à tous ces chiffres ! Il faut faire Math Sup pour afficher une fenêtre ?

Pas d'inquiétude : personne n'est capable de créer un programme informatique ainsi. C'est pourquoi ont été inventés différents langages pour la programmation : le Pascal, le Basic, le C, le C++, le Python, le Java... et bien sûr l'Ada. Ce sont ce que l'on appelle des **langages de haut niveau**. À quoi cela ressemble-t-il ? Voyez vous-même :

Code : Ada

```

with Ada.Text_IO, Ada.Integer_Text_IO ;
use Ada.Text_IO, Ada.Integer_Text_IO ;

procedure exemple is
    n : integer ;
begin
    loop
        Put("Saisir un nombre : ") ;
        Get(n) ; Skip_line ;
        if n mod 2 = 0
            then Put("Ce nombre est pair ! Tres bien !") ; exit ;
            else Put("Ce nombre est impair ! Recommencez. ") ;
        end if ;
    end loop ;
end exemple ;

```

Vous serez capables d'ici quelques chapitres de comprendre ce texte aisément. C'est comme apprendre une langue étrangère ; d'ailleurs si vous connaissez l'anglais, vous avez peut-être reconnu quelques mots : `end`, `if`, `else`, `then`, `begin` et `is`. Ces ainsi que nous crérons nos programmes.

Et comment on traduit ça en 1 et en 0 ?

Une fois notre texte rédigé en Ada (mais c'est aussi valable pour les autres langages), il faudra utiliser un programme-traducteur pour le convertir en un programme lisible par l'ordinateur. Les programmes-traducteurs sont appelés **compilateurs**.

Pourquoi Ada et pas autre chose ?

Il ne vous reste donc plus qu'à apprendre un langage de programmation, et avec ce langage, vous apprendrez également ce que l'on appelle **l'algorithmitique**.

Algorithmique ? Qu'est-ce que c'est que ça encore ?

L'algorithmique est une branche des mathématiques qui traite de la résolution de problèmes à l'aide d'instructions simples. J'en vois déjà certains qui panquent à la vue du mot «mathématiques», mais n'ayez crainte vous avez déjà vu plusieurs fois dans votre vie des algorithmes. Par exemple, comment faire pour diviser 742 par 5 ? Vous vous souvenez, c'était en primaire ? Vous avez appris à vous demander : «dans 7 combien de fois 5 ? Il y va 1 fois et il reste 2». Puis vous répétez cette opération : «dans 24 combien de fois 5 ? Il y va 4 fois et il reste 4...». Il suffisait de répéter plusieurs fois cette méthode basique pour obtenir le résultat désiré (Voir la figure suivante).

$$\begin{array}{r}
 7 \quad 4 \quad 2 \\
 - 5 \quad \downarrow \quad \downarrow \\
 \hline
 2 \quad 4 \quad \downarrow \\
 - 2 \quad 0 \quad \downarrow \\
 \hline
 4 \quad 2 \\
 - 4 \quad 0 \\
 \hline
 2
 \end{array}
 \qquad
 \begin{array}{r}
 5 \\
 \hline
 148
 \end{array}$$

La division illustrée

Eh bien ça, c'était un algorithme : vous répétez des instructions simples dans un certain ordre (abaisser les chiffres de 742, se demander «dans machin, combien de fois true ?»...) afin de résoudre un problème plus compliqué («Combien vaut $742 \div 5$?»). Et tout programme informatique n'est en fait qu'un algorithme écrit dans un langage compréhensible par votre ordinateur. Donc apprendre à programmer, c'est en fait apprendre la science de l'algorithmique dans un langage informatique donné.

Pourquoi Ada ?

Mais alors, pourquoi apprendre Ada et pas simplement l'algorithmique ? Et pourquoi utiliser Ada et pas un autre langage ?

Tout d'abord, l'algorithmique n'est pas compréhensible par les ordinateurs, je vous rappelle que c'est une science. L'algorithme de la division euclidienne vu au-dessus est expliqué en français, or je vous ai dit qu'il doit être écrit dans un langage qui pourra être traduit par un compilateur, un langage très structuré et normé (qui ne changera pas en changeant de compilateur), et ça, c'est ce qu'on appelle la programmation. Et Ada répond à ces exigences.

Au début, on parlait d'Ada tout court ou d'Ada83. Puis le langage a évolué, donnant naissance aux normes Ada95 puis Ada2005 et enfin Ada2012. Mais rassurez-vous, tout cela ne constitue qu'une sorte de «grosse mise à jour» et pas 3 langages différents ! C'est pourquoi nous ne parlerons pas ici des langages Ada83, Ada95 ou Ada2005 mais simplement du langage Ada.

Maintenant, pourquoi utiliser Ada et non pas un autre langage ? Il est vrai que Ada n'est pas le langage le plus répandu. Mais il présente de nombreux avantages par rapport à d'autres langages plus courants comme le C, le C++, le Java...

Tout d'abord il présente un avantage pédagogique. Ada constitue un langage parfait pour apprendre à programmer. D'ailleurs de nombreuses écoles ou universités l'utilisent comme support de leurs cours de programmation. La rigueur qu'il impose au programmeur permet aux débutants d'apprendre à coder correctement, de prendre de bonnes habitudes qu'ils conserveront par la suite, même s'ils venaient à changer de langage de programmation durant leur carrière. Qui plus est, c'est un langage facilement lisible car il utilise des mots complets (`begin`, `end`, `function`, `if`... c'est de l'anglais en effet) plutôt que les symboles «barbares et bizarroïdes» pour un débutant qui préfère la plupart des langages actuels. Les codes des programmes sont certes moins condensés, mais plus faciles à reprendre en cas de bogue ce qui constitue un avantage non négligeable quand on sait que les coûts de maintenance d'un logiciel sont souvent plus élevés que le coût de son développement.

Ensuite, le langage présente également des avantages pour les professionnels. Son fort typage (vous aurez l'occasion de le découvrir durant la deuxième partie) ou sa très bonne gestion des erreurs (abordée lors de la quatrième partie) garantissent des programmes fiables. Ce n'est pas pour rien qu'Ada est utilisé pour de gros projets comme Ariane, chez Thales pour ses avions ou radars, ou encore pour la sécurité des centrales nucléaires, du trafic aérien ou ferroviaire... Utiliser Ada est un gage de fiabilité et pour cause, il a été développé par le polytechnicien français Jean Ichbiah pour le compte du département de la défense américain, le fameux DoD (vous vous doutez bien que rien n'a été laissé au hasard 😊).

Maintenant que ces quelques explications préliminaires sont faites, j'espère que vous vous sentez toujours prêts à vouloir apprendre un langage. Si cela vous semble compliqué, n'ayez crainte, nous commencerons doucement et je vous guiderai pas à pas. Nos premiers programmes seront peut-être inutiles ou ringards, mais nous apprendrons peu à peu de nouvelles notions qui nous permettront de les perfectionner.

En résumé :

- Un programme est un fichier contenant des instructions à destination du processeur de votre ordinateur.
- Un programme est écrit en **langage binaire**, le seul langage que comprend votre ordinateur.
- Pour réaliser un programme, vous devez écrire des instructions dans un fichier texte. Celles-ci sont écrites dans un **langage de programmation** (ou langage formel) comme Ada, compréhensible par les humains.
- Pour traduire un texte écrit en langage de programmation en un programme écrit en binaire, vous devrez utiliser un logiciel appelé **compilateur**.

Les logiciels nécessaires

Nous allons au cours de ce chapitre, télécharger et installer les logiciels nécessaires à l'utilisation d'Ada.



Eh bien non. Comme tout langage, Ada a besoin de deux types de logiciels : un éditeur de texte avancé (appelé IDE) et un compilateur. Nous utiliserons l'IDE appelé Adagide (même si le logiciel GPS pourra être utilisé) et le compilateur GNAT. Après quoi, nous pourrons découvrir (enfin) le joyeux monde de la programmation.

IDE et compilateur : kesako ?

Le compilateur

Nous avons donc et déjà parlé du compilateur : il s'agit d'un programme qui va traduire notre langage Ada en un vrai programme utilisable par l'ordinateur. Le compilateur est aussi chargé de vérifier la syntaxe de notre code : est-il bien écrit ? Sans fautes d'orthographe ou de grammaire ? Il existe plusieurs compilateurs en Ada, mais assurez-vous, quel que soit celui que vous choisissez, les règles sur le langage Ada seront toujours les mêmes car Ada est un langage nommé (il bénéficie d'une norme internationale assurant que tous les compilateurs Ada respecteront les mêmes règles). Il existe plusieurs compilateurs Ada (vous en trouverez une liste à l'adresse suivante), mais la plupart sont des programmes payants et propriétaires (donc non libre).



Rassurez-vous, nous allons plutôt opter pour un compilateur libre et gratuit. Ce compilateur s'appelle GNAT et est développé par la société Adacore. Il a longtemps été payant mais une version gratuite existe désormais (GNAT GPL).

L'IDE ou EDI

Cela dit, le compilateur n'est en définitive qu'un «traducteur». Il ne créera pas de programme à votre place, c'est-à-vous d'écrire les instructions dans des fichiers textes. Pour cela, vous pouvez écrire vos documents sous l'éditeur de texte de votre ordinateur (notepad sous Windows par exemple), ou utiliser un éditeur de texte un peu plus avancé comme Notepad++ qui colorera votre texte intelligemment en reconnaissant le langage dans lequel vous programmez (on appelle ça la coloration syntaxique).

Mais le plus simple, c'est d'utiliser un Environnement de Développement Intégré, EDI en français ou IDE en anglais. Pourquoi ? Eh bien parce que ce logiciel vous fournira un éditeur de texte avec coloration syntaxique et surtout vous permettra de compiler votre texte aisément, sans avoir besoin de chercher où se trouve le compilateur sur votre ordinateur. De plus, l'IDE se chargera de le paramétrier pour vous et vous fournira divers outils utiles à la programmation.

L'IDE que nous allons utiliser s'appelle Adagide et il est gratuit.

Télécharger et installer Adagide.

Disponible sous Windows uniquement

Téléchargement

Cliquez [ici](#) pour télécharger Adagide dans sa version 7.45 (dernière en date à l'heure où j'écris ces lignes). Puis cliquez sur «adagide-7.45-setup.exe» pour lancer le téléchargement de l'installateur.

Installation

L'installation d'Adagide ne devrait pas vous poser de problèmes. Il vous suffit de suivre les différentes étapes des figures suivantes en cliquant sur «Suivant». Acceptez, tout d'abord, les termes de la licence et ensuite, à l'écran «Setup Type», choisissez «Typical».



Télécharger et installer GNAT.

Disponible sous Windows et Linux

Téléchargement

Pour télécharger le compilateur GNAT GPL, rendez-vous sur le site d'Adacore en cliquant [ici](#).

Si vous êtes sous Windows, vérifiez que le système d'exploitation est bien «x86 Windows» (à la ligne «Select your platform»), puis cliquez sur «GNAT GPL» et sélectionnez «gnat-gpl-2013-i686-pc-mingw32-bin.exe». Enfin, tout en bas de la page, cliquez sur le bouton «Download selected files».

Pour les utilisateurs de Linux, le système d'exploitation devrait être «x86 - Linux» (ou «x86_64 - Linux» si votre processeur est en 64 bits). De la même manière que sous Windows, cliquez sur «GNAT GPL» et sélectionnez «gnat-gpl-2013-i686-gnu-linux-libc2.3-bin.targz». Pour finir, cliquez sur le bouton «Download selected files» en bas de page.

Installation

L'installation est un poil plus compliquée (même si ça ne casse pas trois pattes à un canard, comme dirait ma grand-mère). Le fichier téléchargé s'appelle «AdaCore.tar» et ce n'est pas un programme d'installation, seulement un fichier compressé. Pour le décompresser, utilisez un logiciel comme Winrar ou 7zip. Sous Linux, vous pouvez procéder graphiquement ou par la console. Dans le second cas, vous devrez taper la commande suivante :

Code : Console

```
tar -xvzf gnat-gpl-2013-i686-gnu-linux-libc2.3-bin.tar.gz
```

Sous Windows, vous obtiendrez ainsi le fameux fichier d'installation "gnat-gpl-2013-i686-pc-mingw32-bin.exe". Ouvrez-le et suivez les différentes étapes. Je vous conseille de sélectionner l'option «*Install for all users*», notamment si votre ordinateur dispose de plusieurs sessions, à moins que vous ne vouliez être le seul à pouvoir programmer en Ada.

Sous Linux, l'installation se fera via le fichier script appelé «*doinstall*». Celui-ci vous demandera si vous souhaitez installer GNAT et si oui, où. Vous n'aurez qu'à appuyer deux fois sur «Entrée» puis répondre «Yes» aux deux dernières questions en appuyant sur les touches Y ou y.

Télécharger et installer GPS

GPS est un autre IDE, mais développé par AdaCore. Il est un peu plus compliqué et j'avoue avoir eu du mal à l'installer sur mon PC Windows personnel (même si il fonctionne sur mon PC Windows de bureau ou sous mon PC Ubuntu ). GPS est conçu pour gérer des projets importants composés de nombreux fichiers, mais Adagide a l'avantage d'être simple et performant, je le préférerais donc à GPS pour ce cours. Mais si vous souhaitez tout de même utiliser GPS, sachez qu'à cette étape du chapitre... il est déjà installé !  Cet IDE est en effet livré avec GNAT, vous pourrez donc le tester si la curiosité vous prend : il suffit d'aller dans le menu Démarrer > Programmes > GNAT > 2010 > GPS.



Si vous êtes sous Linux, vous ne pouvez disposer d'Adagide. Vous serez donc amenés à utiliser l'IDE d'AdaCore : GPS. Si celui-ci n'apparaît pas dans vos applications, vous pourrez le lancer en tapant la commande «*gnat-gps*» dans votre console ou bien en l'ajoutant dans la liste de vos applications.

En résumé :

- Vous devez disposer d'un IDE ou, à défaut, d'un éditeur de texte afin de rédiger le code source de votre programme.
- Je vous conseille d'utiliser l'IDE Adagide pour sa simplicité. N'optez pour GPS que lorsque vous aurez acquis la maturité suffisante.
- Vous devez disposer d'une version du compilateur GNAT.

Notre premier programme en Ada

Bon, nous avons installé notre IDE, Adagide, notre compilateur, GNAT. Je sens que vous commencez à perdre patience. Nous allons immédiatement pouvoir nous atteler à notre premier programme en Ada. Il s'agira de créer un programme qui nous salut. Bon, ce n'est sûrement pas folichon comme objectif, mais c'est un début. Voyons ici la démarche que nous allons suivre.

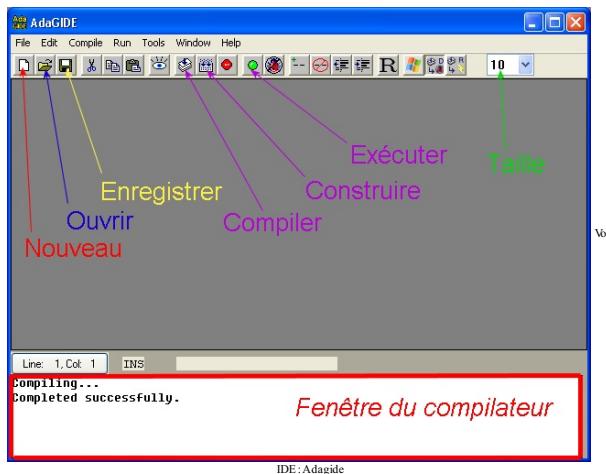
1. Tout d'abord, nous allons faire un petit tour d'horizon des logiciels Adagide et GPS. Je vous rassure, ce sera rapide.
2. Puis nous écrirons notre premier programme et nous le testerons.
3. Enfin, nous essaierons de décompiler notre premier programme.
4. Avant de commencer le prochain chapitre je vous proposerai quelques petits suppléments et exercices pour vous entraîner.

Toujours pas découragé ? Alors au travail !

Découvrir son IDE en quelques secondes

Soyons rapide avec Adagide

Pour pouvoir programmer, nous avons déjà expliqué que nous aurons besoin de l'IDE Adagide. Donc, lancez Adagide ! Vous voilà donc face à une fenêtre (voir la figure suivante) tout ce qu'il y a de plus... austère. Mais cela n'a que peu d'importance pour nous.



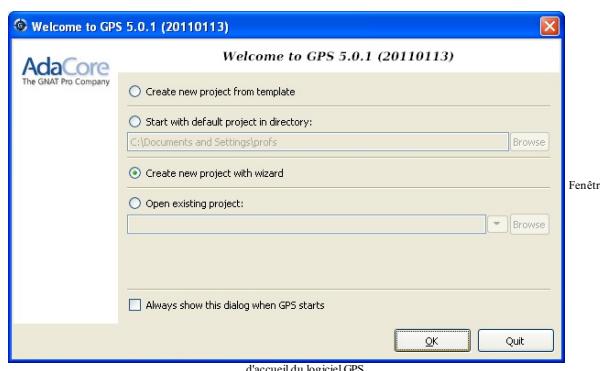
Tout d'abord, nous allons créer un nouveau document. Pour cela, cliquez sur *File > New* ou sur l'icône «Nouveau» indiqué sur la figure précédente. Vous pourrez également à l'avenir ouvrir un document existant en cliquant sur *File > Open* ou sur l'icône «Ouvrir».

Pour l'heure, nous allons enregistrer notre document. Cliquez sur *File > Save as* ou sur l'icône «enregistrer». Je vous conseille de créer un répertoire que nous nommerons «Hello», et dans ce répertoire nous allons enregistrer notre document (appelons-le également «Hello»). Vous remarquerez que l'extension proposée est .adb. Notre code en Ada portera le nom de Hello.adb ; par la suite nous verrons à quoi servir l'extension .ads qui est également proposée.

Bien entendu, il est possible d'écrire dans la fenêtre principale, la taille du texte pouvant être modifiée en cliquant sur le bouton «taille». En revanche, il est impossible d'écrire dans la partie basse de la fenêtre, celle-ci étant réservée au compilateur. C'est ici que le compilateur affichera les informations qui vous seront nécessaires : échec de la compilation, réussite de la compilation, lignes inutiles ...

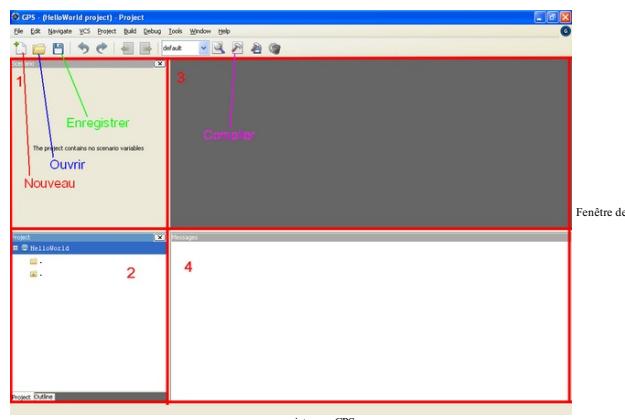
Pour les utilisateurs de GPS (plus long)

Le logiciel GPS est un peu plus compliqué que Adagide. Il est toutefois mieux adapté à de gros projets, ce qui explique qu'il soit plus complexe et donc moins adapté pour ce cours. Lorsque vous lancez GPS, une fenêtre devrait vous demander ce que vous souhaitez faire.



d'accueil du logiciel GPS

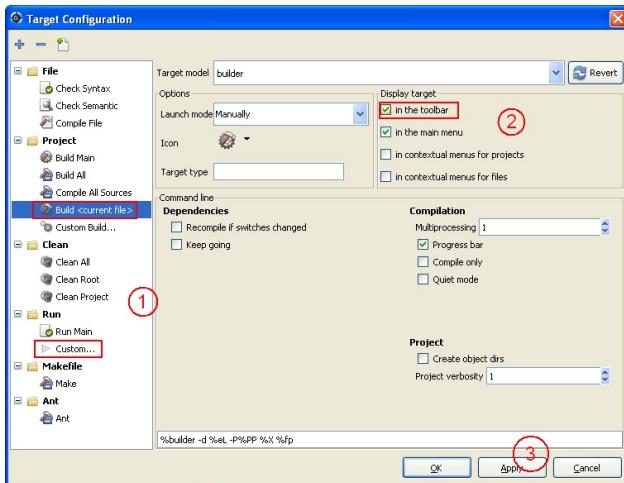
Pour l'heure, choisissez «Create new project with wizard» et cliquez sur «OK». Choisissez l'option «Single project» puis cliquez sur «Forward». Choisissez un nom pour votre projet (j'ai choisi «HelloWorld») et précisez un répertoire où l'enregistrer (je vous conseille de créer un répertoire spécifique par projet). Cliquez ensuite sur «Apply» sans renseigner les autres informations. Vous devriez arriver à l'écran suivant :



La fenêtre est découpée en quatre zones. La première ne nous intéresse pas pour l'instant. La seconde, juste en dessous, affichera les fichiers et programmes de votre projet. Pour l'heure vous n'aurez qu'un seul fichier donc cela ne comportera pas d'intérêt pour les premiers cours. Son utilité se révélera à partir de la Partie III. La troisième zone de la fenêtre est la zone principale, celle où vous rédigez votre programme. Enfin, la quatrième zone est celle réservée au compilateur et à la console. Vous comprendrez son utilité bientôt.

Commencez donc par créer un nouveau fichier en cliquant sur l'icône «Nouveau» ou sur *File > New*. Sauvegardez tout de suite votre fichier en cliquant soit sur l'icône «Enregistrer» soit sur *File > Save As*. Votre fichier devra s'appeler «Hello.adb» (même si vous n'avez pas besoin d'écrire l'extension). Comme je vous le disais plus haut, il existe également un fichier *ads* que nous verrons plus tard et que le logiciel GPS gère aussi. Pour ouvrir un document, vous pourrez utiliser l'icône «Ouvrir» ou sur *File > Open*.

Voilà pour l'essentiel, toutefois, je vais vous demander d'effectuer deux petites manipulations avant d'aller plus loin afin d'ajouter deux icônes importantes. Cliquez sur le menu *Build > Settings > Targets*. Une fenêtre devrait s'ouvrir (voir la figure suivante).



Fenêtre de configuration

À gauche, dans la section *Project*, cliquez sur *Build <current file>* et cochez la case *In the toolbar*. Puis, dans la section *Run*, cliquez sur *Custom* et cochez la case *In the toolbar*. Enfin, cliquez sur le bouton *Apply*. Deux icônes devraient s'ajouter, comme celles des figures suivantes (retenez-les bien).



Notre premier programme Un petit copier-coller !

Maintenant que nous avons rapidement pris connaissance du logiciel, il est temps de nous lancer corps et âme dans la création de notre magnifique programme «Hello» ! Nous allons pour cela effectuer une opération de haute voltige : un copier-coller ! Voici le code d'un programme. Ne cherchez pas à comprendre pour l'instant, je vous expliquerai par la suite. Pour l'heure, sélectionnez le texte, copiez-le et collez-le dans votre fichier Hello.adb sans poser de questions.

```
Code : Ada
with ada.text_io;
use ada.text_io;

procedure Hello is
    --partie réservée aux déclarations
begin
    put("Salut tout le monde !"); --on affiche un message
end Hello;
```

Vous remarquerez que certains mots sont automatiquement colorés, je vous en ai déjà parlé, c'est la coloration syntaxique. Elle permet de faire ressortir certains mots-clés (Adagide colore par défaut en bleu les mots réservés au langage Ada) ou certains types de texte.

Avant d'aller plus loin, si vous êtes sous Adagide (GPS effectuera cette manœuvre tout seul le moment venu), il serait bon de cliquer sur l'icône «Reformat», il est simple à trouver, c'est celui avec un grand R dessiné dessus. Cela va mettre votre code en forme : la ligne entre **is** et **begin** et la ligne entre **begin** et **end** vont être «avancées» à l'aide d'une tabulation (trois espaces sous adagide). On appelle ça l'**indentation**. Ca n'a rien de rien mais c'est très important car à l'avenir votre code pourra s'étendre sur plusieurs pages, il est donc important qu'il soit le plus lisible possible. Lorsque l'on rédige un roman, on crée des paragraphes, des chapitres... en programmation on indente son texte.

Compiler, créer... lancer !

Maintenant que votre texte est écrit et mis en forme, il est temps de le faire fonctionner. Nous allons donc utiliser le compilateur. Pour cela, rien de plus simple, cliquez sur l'icône «Compiler» (ou appuyez sur F2 avec Adagide, ou encore *Compile > Compile File*). Le compilateur vérifiera la syntaxe de votre code. Si il n'y a pas d'erreur (et il ne devrait pas y en avoir), le compilateur devrait vous indiquer (dans la fenêtre du bas) «**Completed successfully**.

Mais à ce stade, votre programme n'existe pas encore, vous devez construire l'exécutable. Pour cela cliquez sur l'icône «**Exécuter**» (ou appuyez sur F3 avec Adagide ou cliquez sur *Compile > Build*).

Pour les utilisateurs de GPS, les manipulations étant un peu longues, je vous conseille d'utiliser les icônes (c'est bien pour cela que je vous ai fait faire une petite manipulation préliminaire). Lorsque vous cliquerez sur l'icône «**Construire**», une fenêtre peut s'ouvrir. Ne vous souciez pas des paramètres et cliquez sur OK.

Votre fichier exécutable est désormais créé. Vous pouvez soit aller le chercher dans le répertoire Hello que vous avez créé tout à l'heure, soit cliquer sur l'icône «**Exécuter**» (ou appuyer sur F4 avec Adagide ou cliquer sur *Run > Execute*). Sous GPS, une fenêtre s'ouvrira. Ne cochez aucune case et, si la barre de texte est vide, écrivez-y le nom de votre programme : Hello (sans extension !). Puis cliquez sur OK.

Avec Adagide, vous devriez ainsi obtenir une magnifique fenêtre noire vous indiquant :

```
Code : Console
Salut tout le monde !
```

(Sous GPS, ces actions apparaîtront dans la fenêtre du compilateur, c'est-à-dire la quatrième zone de la fenêtre)



Ne perdez pas patience. Le chemin sera long avant que vous ne puissiez créer un programme digne d'intérêt. 🐸 Prends le temps de décortiquer ce que nous avons copié pour mieux comprendre.

Mon dieu, qu'ai-je fait ?

Si nous jetons un œil à notre code nous pouvons nous rendre compte qu'il peut se décomposer en deux parties majeures : une sorte de titre (avec **with** et **use**) et un gros paragraphe (commençant par le mot **procedure**).

		Titre		
		Partie n°1	Partie n°2	Paragraphe
			Organisation de votre code	

Le corps du programme : la procédure Hello

Le titre étant un peu compliqué à expliquer, nous allons commencer par nous intéresser au « gros paragraphe » : la procédure appelée Hello. Elle peut se décomposer elle-même en deux parties comme sur la figure précédente.

Les trois bornes de la procédure

Pour l'instant, disons que le terme de «procédure» est un synonyme de «programme». Notre paragraphe commence donc par l'introduction suivante : **PROCEDURE Hello IS**. Cette phrase permet de donner un nom à notre procédure et indique le début du texte le concernant. Remarquez que les mots **procedure** et **is** sont colorés en bleu : ce sont des mots réservés par le langage Ada, ils ont donc un sens très précis et ne peuvent être utilisés n'importe comment.

Deuxième «borne» de notre procédure : le terme **begin**. C'est lui aussi un mot réservé. Il indique au compilateur le début des instructions que l'ordinateur devra exécuter.

Troisième «borne» : le mot **end**, ou plus exactement la phrase : «**END Hello ;** ». Cette phrase indique au compilateur la fin de la procédure Hello. À noter que contrairement aux deux bornes précédentes, cette phrase se termine par un point-virgule. Si vous oubliez de l'écrire, le compilateur vous avertira : **missing ";"** ! Notez que les points virgule indiquent au compilateur la fin d'une instruction : ici, c'est la fin de votre procédure.

La partie Déclaration

Les trois bornes vues précédemment délimitent deux zones. La première, celle comprise entre **is** et **begin**, est réservée aux déclarations. Nous verrons dans les prochains chapitres de quoi il retourne. Sachez pour l'instant que votre programme peut avoir besoin de mémoire supplémentaire pour fonctionner (et nous aurons très vite besoin de davantage de mémoire) et que c'est dans cette partie que s'effectueront les demandes de réquisition de mémoire. Pour l'heure, il n'y a rien.

Comment ça il n'y a rien ? Il y a bien quelque chose d'écrit : «--partie réservée aux déclarations»

Eh bien non. Pour le compilateur, il n'y a rien ! ☺ Cette ligne verte commence par deux tиртs. Cela signifie qu'il s'agit d'un commentaire, c'est-à-dire d'un texte qui ne sera pas pris en compte par le compilateur. Quel intérêt d'écrire du texte s'il n'est pas pris en compte par le compilateur ? Eh bien cela permet d'apporter des annotations à votre code. Si vous relisez un code écrit par quelqu'un d'autre ou par vous même il y a longtemps, vous risquez d'avoir du mal à le comprendre. Les commentaires sont donc là pour expliquer ce que fait le programme, à quoi s'servent telle ou telle ligne, etc. Un bon code est déjà un code bien commenté (et bien indenté également).

La partie Action et notre première instruction

Puis, après le **begin**, vient la partie la plus intéressante, celle où l'on indique au programme ce qu'il doit faire, autrement dit, c'est là que nous écrivons les différentes instructions.

L'instruction citée ici est : «**Put ("Salut tout le monde!")** ; ». C'est assez simple à comprendre : l'instruction **Put()** indique à l'ordinateur qu'il doit afficher quelque chose. Et entre les parenthèses, on indique ce qu'il faut afficher. Il est possible d'afficher un nombre (par exemple, «**Put(13)** ; » affichera le nombre 13) comme du texte, mais le texte doit être écrit entre guillemets.

Remarquez là encore que l'instruction se termine par un point-virgule. D'ailleurs, toutes les instructions devront se terminer par un point-virgule, à quelques exceptions près (souvenez-vous de **IS** et **BEGIN**). Le texte qui suit est bien entendu un commentaire et tout ce qui suit les double-tиртs ne sera pas pris en compte par le compilateur.

Il est bien sûr possible d'afficher autre chose que "salut tout le monde !" ou d'effectuer d'autres actions. Toutefois, lorsque toutes vos actions ont été écrites, n'oubliez pas d'indiquer au compilateur que vous avez atteint la fin du programme en utilisant le mot-clé **END**.

Les Packages avec With et Use

Le mot-clé WITH

Revenons maintenant au titre. Pourquoi cet intitulé ? Et d'ailleurs pourquoi l'écrire deux fois ? Je vous propose de le supprimer pour mieux comprendre. Compiler à nouveau votre code.

Argh !! Ça ne marche plus ! J'ai plein de messages rouge ! Tout est fichu !

En effet, le code n'est plus correct et ce n'est pas la peine d'essayer de reconstruire notre programme. Mais lisons tout d'abord les avertissements du compilateur : «**Put is undefined.**

Cela signifie tout simplement que le compilateur ne connaît plus l'instruction **Put()** ! En effet, le compilateur ne connaît que très peu de mots et d'instructions : les mots réservés et puis, c'est presque tout. Pour aller plus loin, il a besoin de fichiers qui contiennent davantage d'instructions comme l'instruction **Put()** par exemple, qui se situe dans un fichier appelé **Ada.Text_IO**. Ces fichiers portent le nom de **package** en Ada (paquetages en français). Dans d'autres langages, on parle de bibliothèques.

Nous reviendrons plus en détail sur les packages durant la troisième et la quatrième partie de ce tutoriel. Donc si cette sous-partie vous semble compliquée, rassurez-vous, je vous indiquerai toujours les packages à écrire en début de code.

Le compilateur nous dit également : **possible missing <WITH Ada.Text_IO ; USE Ada.Text_IO >**.

Traduction : ce serait bien de remettre les lignes que vous avez supprimé tout à l'heure ! Écoutons-le mais ne réécrivons que la première ligne (au tout début) :

Code : Ada

```
WITH Ada.Text_IO ;
```

Le mot-clé USE

Réessayons de compiler.

J'ai un nouveau problème. Le compilateur m'indique : «**Put is not visible**, plus plein d'autres insutes incompréhensibles. Apparemment, il connaît mon instruction Put() mais il n'arrive plus à la voir ?!

Exactement. Et les lignes qui suivent indiquent que dans le package **ada.Text.IO**, il y a plusieurs références à l'instruction **Put()**. En fait, il est possible après l'instruction **with** d'écrire de nombreux packages, autant que vous voulez même (nous verrons un exemple juste après). Mais il peut exister (et il existe) plusieurs instructions portant le nom **Put()**, du coup le compilateur ne sait pas d'où vient cette instruction. De laquelle s'agit-il ? Une solution est de remplacer **Put()** par **Ada.Text.IO.Put()** ! C'est compliqué, c'est long à écrire et nous aurons tout le loisir de comprendre tout cela plus tard. Donc une façon de s'épargner ces difficultés, c'est d'écrire notre instruction **use** au tout début, juste après l'instruction **with**.

Ces deux lignes (appelées Context Clause ou Clauses de contexte en Français) sont donc indispensables et vous serez souvent amenés à réécrire les mêmes (vous bénirez bientôt l'inventeur du copier-coller). Ne craindez pas que cette lourdeur soit spécifique à Ada. Tout langage a besoin de packages ou bibliothèques annexes, tout n'est pas prédefini et heureusement pour nous : c'est ce que l'on appelle la **modularité**. Au final, voici la structure de notre programme et de tout programme Ada (voir la figure)

		Packages	
		Partie déclarations	Procédure
		Partie Instructions	
			La structure du programme

Avec plusieurs packages

J'ai essayé de bidouiller le code pour qu'il affiche 13 comme tu le disais dans un exemple. Mais le compilateur râle encore : **warning : no entities of <Text_Io> are referenced, no candidate match the actuals** ... Bref, rien ne va plus.

 Faut-il un autre package ?

Exactement ! Ada.Text_IO est un package créé pour gérer le texte comme son nom l'indique (Text = texte et IO = In Out, entrée et sortie). Pour afficher un nombre entier (ici 13), il faut utiliser le package Ada.Integer_Text_IO.

Par exemple :

Code : Ada

```
WITH Ada.Text_IO,Ada.Integer_Text_IO ;
USE Ada.Text_IO,Ada.Integer_Text_IO ;
```



Les instructions **WITH** et **USE** sont terminées par un point-virgule, en revanche les deux packages sont simplement séparés par une virgule !

Pour plus de clareté, il est également possible d'écrire ceci :

Code : Ada

```
WITH Ada.Text_IO,
     Ada.Integer_Text_IO ;
USE Ada.Text_IO,
     Ada.Integer_Text_IO ;
```

ou ceci :

Code : Ada

```
WITH Ada.Text_IO ;           USE Ada.Text_IO ;
WITH Ada.Integer_Text_IO ;   USE Ada.Integer_Text_IO ;
```

Et oui ! Bien Présenter son code, c'est très important !

Une dernière remarque qui a son importance

Vous devez savoir avant de vous lancer à l'aventure que le langage Ada n'est pas «**case sensitive**», comme diraient nos amis anglais. Cela signifie qu'il ne tient pas compte de la casse. Autrement dit, que vous écrivez en majuscule ou en minuscule, cela revient au même : **BEGIN**, **begin** ou **bEGIN** seront compris de la même manière par le compilateur. Toutefois, Ada sait faire la différence lorsqu'il faut. Si vous modifiez la casse dans la phrase à afficher "Salut tout le monde !", l'affichage en sera modifié.

De même, une tabulation ou un espace seront pris en compte de la même façon. Et que vous tapiez un, deux ou trois espaces ne changera rien.

Exercices

Exercice 1

Ce premier programme est peut-être simple, mais voici quelques exercices pour nous amuser encore. Nous avons vu l'instruction Put() qui affiche du texte, en voici maintenant une nouvelle : New_line. Elle permet de retourner à la ligne. N'oubliez pas le point virgule à la fin.

Énoncé

Écrire un programme affichant ceci :

Code : Console

```
Coucou
tout le
monde
! ! !
```

Solution

Secret (cliquez pour afficher)

Code : Ada

```
With ada.text_io ;
use ada.text_io ;

procedure Hello2 is
begin
  Put("Coucou") ; New_line ;
  Put("tout le") ; New_line ;
  Put("monde") ; New_line ;
  Put("! !") ;
end Hello2 ;
```

Exercice 2

Troisième instruction : Put_line(). Elle fonctionne comme Put(), sauf qu'elle crée automatiquement un retour à la ligne à la fin.

Énoncé

Même exercice que précédemment, mais sans utiliser New_line.

Solution

Secret (cliquez pour afficher)

Code : Ada

```
With ada.text_io ;
use ada.text_io ;

procedure Hello3 is
begin
  Put_line("Coucou") ;
  Put_line("tout le") ;
  Put_line("monde") ;
  Put("! !") ;
  -- pas besoin de put_line ici, on est arrivé à la fin
end Hello3 ;
```

Exercice 3

Énoncé

Pourquoi ne pas créer des programmes affichant autre chose que "coucou tout le monde !".



Attention, vous n'avez pas le droit aux caractères accentués. Uniquement les caractères anglo-saxons si vous ne voulez pas avoir de drôles de surprises. Donc pas de è, è, à, à, ê, ê, ô...

Vous avez désormais créé votre premier programme. Certes, il n'est pas révolutionnaire, mais c'est un début. Nous allons pouvoir quitter la première partie de ce tutoriel et entrer dans les bases de la programmation en Ada. La prochaine partie nous permettra de manipuler des nombres, d'effectuer des opérations, de saisir des valeurs au clavier... peu à peu nous allons apprendre à créer des programmes de plus en plus complexes.

En résumé :

- Tout programme débute par les clauses de contexte : à l'aide des mots clés **WITH** et **USE**, vous devez lister les packages nécessaires.
- Les instructions Put(), Put_Line() et New_Line() sont accessibles avec le package Ada.Text_IO. Elles permettent respectivement d'écrire du texte, d'écrire une ligne et de retourner à la ligne.
- Les instructions doivent s'achever par un point virgule.
- Prenez soin de bien présenter votre code. Pour cela, indentez-le en utilisant la tabulation et commentez-le en écrivant quelques indication après un double-tiret.
- Les mots réservés, comme **BEGIN**, **END**, **PROCEDURE**, **WITH** ou **USE**, ont un sens bien précis et ne pourront pas être utilisés pour autre chose que ce pour quoi ils sont prévus.

Maintenant que Adagide et GNAT sont installés et que nous avons une idée plus précise de ce qui nous attend, nous allons pouvoir commencer notre apprentissage du langage Ada.

Partie 2 : Ada, notions essentielles

Nous allons voir dans cette partie, les éléments essentiels du langage Ada et de tout langage de programmation. Nous allons nous intéresser à la grammaire de base du langage. Au menu : les variables (typage, affectation et opérations), les conditions, les boucles, les procédures et les fonctions. Tout cela constitue le B.A.BA de tout programmeur et vous ne pourrez pas en faire l'impasse. Enfin, nous terminerons par un premier TP : la création d'un petit jeu de dés en mode console, le craps. Mais avant d'en arriver là, nous avons du pain sur la planche !

Variables I : Typage et affectation

Nous avons donc déjà réalisé notre premier programme. Mais chacun comprend vite les limites de notre programme "Hello". À dire vrai, un tel programme n'apporte absolument rien. Ce qui serait intéressant, ce serait que l'utilisateur de notre programme puisse entrer des informations que l'ordinateur lui demanderait.

Par exemple :

Code : Console

Quel âge avez-vous ? _

Et là nous pourrions entrer 25, 37 ou 71. Le programme pourrait ensuite se charger d'enregistrer cette information dans un fichier, de nous avertir si nous sommes considérés comme majeur dans l'état du Minnesota ou encore de nous donner notre âge en 2050... Autant d'applications que nous ne pourrons pas réaliser si l'âge que l'utilisateur indiquera n'est pas enregistré dans une zone de la mémoire.

Et c'est là que les variables interviennent ! Attention, ce chapitre n'est peut-être pas exaltant mais il est **absolument nécessaire** à la compréhension des prochains.

Déclaration de variables

Nous allons reprendre l'exemple de l'introduction. Nous voulons indiquer un âge à notre programme. Nous aurons donc besoin d'un espace mémoire capable de stocker un nombre entier. Il faudrait effectuer une demande d'allocation d'une zone mémoire et retenir l'adresse mémoire qui nous a été attribuée et la taille de la zone allouée.

Euh... c'était pas marqué Niveau facile ? 😊

Pas d'inquiétude. Tout cela, c'est ce qu'il aurait fallu faire (entre autre) si nous n'avions pas utilisé un langage comme Ada. C'est ce qu'on appelle un langage de haut niveau, c'est-à-dire que l'on n'a pas besoin de se casser la tête pour faire tout cela. Il suffit de dire « Je veux de la mémoire ».

Toutefois, l'ordinateur a tout de même besoin de connaître la place mémoire dont vous aurez besoin et cela dépend du **type** d'information que l'on souhaite y enregistrer. Ici, nous voulons enregistrer un nombre entier que nous appellerons *age*. Le mot *age* est le nom de notre variable, c'est-à-dire l'emplacement de la mémoire où seront enregistrées nos informations. Pour faire tout cela, il faut **déclarer** notre variable (comme on le ferait à la douane, cela permet de savoir qui vous êtes et comment vous retrouver). Une déclaration se fait toujours de la façon suivante :

NOM_DE_LA_VARIABLE : TYPE ;

Il est possible de déclarer plusieurs variables d'un coup de la manière suivante :

NOM_DE_LA_VARIABLE1 , NOM_DE_LA_VARIABLE2 : TYPE ;

Il est aussi possible, aussitôt la variable déclarée, de lui affecter une valeur à l'aide du symbole « := ». On écrit alors :

NOM_DE_LA_VARIABLE : TYPE := VALEUR ;

Où dois-je faire cette déclaration ? En préfecture ? 😊

Vous vous souvenez notre premier programme ? Je vous avais parlé d'une zone réservée aux déclarations (entre **is** et **begin**). Et bien c'est ici que nous déclarerons notre variable.

Code : Ada

```
Procedure VotreAge is
    NOM_DE_LA_VARIABLE1 : TYPE1 := VALEUR ; --
    zone réservée aux déclarations
    NOM_DE_LA_VARIABLE2 : TYPE2 ;
begin
    Put("Quel age avez-vous ?") ;
end ;
```

Bien, il est temps de rentrer dans le vif du sujet : comment déclarer notre variable âge ?

Définition

Le type **Integer** est réservé aux nombres *entiers relatifs*. Pour ceux qui ne se souviendraient pas de leur cours de mathématiques, un *entier* est un nombre qui n'a pas de chiffres après la virgule à part 0. *Relatif* signifie que ces entiers peuvent être positifs (avec un signe +) ou négatifs (avec un signe -).

Le type **Natural** est réservé aux nombres entiers naturels (c'est-à-dire positifs ou nul). Il est donc impossible, si vous déclarez votre variable comme un **Natural**, que celle-ci soit négative (ou alors vous planterez votre programme). Pour être plus précis, c'est un sous-type de type **Integer**. Quelle importance ? Eh bien, cela signifie que l'on pourra «mélanger» les natural et les Integer sans risquer le plantage : nous pourrons les additionner entre eux, les soustraire, les multiplier... ce qui n'est pas possible avec d'autres types. La seule restriction est que notre **Natural** ne devienne pas négatif.

Valeurs possibles

Si **N** est une variable de type **Integer**, elle peut prendre les valeurs suivantes :

0 ; 1 ; 2 ; 3 ; 4... 2 147 483 647
-1 ; -2 ; -3... - 2 147 483 648



On remarquera qu'il y a moins de positifs que de négatifs. Cela tient à une raison toute simple, c'est que 0 est compté comme un positif et pas comme un négatif. Comme le nombre d'octets nécessaires à l'enregistrement d'un nombre est limité (ici 31 bits pour la partie numérique et 1 bit pour le signe soit 32 bits = 4 octets), cela implique que l'on peut aller «moins loin dans les positifs que dans les négatifs».

Si **N** est une variable de type **Natural**, elle peut prendre les valeurs suivantes :

0 ; 1 ; 2 ; 3 ... 2 147 483 648

On remarque qu'il ne s'agit là que d'une restriction du type **Integer**. Comme je vous l'ai dit, **Natural** est un sous-type de **Integer** (noté **subtype** en Ada). Il n'y a donc pas davantage de **Natural** que d'**Integer** positifs ou nuls.



Il existe également un type **Positive**, semblable aux **Natural** mais pour lequel 0 est exclu.

Exemple

Nous souhaiterions créer un programme qui affiche votre âge. Au début de notre code, nous devrons donc écrire :

Code : Ada

```
Procedure VotreAge is
    age : Integer := 27;
begin
    ...
```

Ou bien, sans affecter de valeur :

Code : Ada

```
Procedure VotreAge is
    age : Integer ;
begin
...
```

Si nous voulons pouvoir afficher des Integer ou des Natural, il faudra ajouter Ada.Integer_Text_IO dans la liste des packages. Ce package contient une instruction Put() prévue spécifiquement pour les nombres entiers. Attention, il existe donc deux instructions Put() différentes : une pour le texte et une pour les entiers ! Une dans le package Ada.Text_IO et une dans Ada.Integer_Text_IO. Réalisons maintenant ledit programme et voyons le résultat :

Code : Ada

```
WITH Ada.Text_IO ;           USE Ada.Text_IO
WITH Ada.Integer_Text_IO ;   USE Ada.Integer_Text_IO ;

PROCEDURE VotreAge IS
    age : Integer := 27 ;
BEGIN
    Put("Vous avez ");
    Put(age) ;
    Put(" ans.") ;
END VotreAge ;
```

Code : Console

```
Vous avez      27 ans.
```



Par défaut, l'instruction Ada.Integer_Text_IO.Put() (c'est son nom complet) réserve toujours la même place pour afficher des entiers, qu'ils soient petits ou grands. Mais il est possible de spécifier la taille de cet emplacement. Pour cela, il y a possibilité d'ajouter des « options » à notre instruction Put() (on parle plus exactement de paramètres). Juste après age, il vous suffit d'écrire une virgule suivie de « Width => ### ». Vous remplacerez les dièses par le nombre de chiffres à afficher. Ce paramètre Width est un mot anglais qui signifie « Largeur ». Par exemple :

Code : Ada

```
WITH Ada.Text_IO ;           USE Ada.Text_IO
WITH Ada.Integer_Text_IO ;   USE Ada.Integer_Text_IO ;

PROCEDURE VotreAge IS
    age : Integer := 27 ;
BEGIN
    Put("Vous avez ");
    Put(age, Width => 0) ;
    Put(" ans.") ;
END VotreAge ;
```

En inscrivant « Width => 0 », on indique que l'on ne veut afficher aucun chiffre. Mais le langage Ada étant bien conçu, il affichera tout de même les chiffres réellement utiles, mais pas un de plus ! Cé qui nous donnera :

Code : Console

```
Vous avez 27 ans.
```

Le type Float

Définition

Le type float est chargé de représenter les nombres décimaux. Dans les faits, il n'en représente en fait qu'une partie (comme les Integer ne couvrent pas tous les nombres entiers).



Autant Natural = Naturel ; Integer = Entier, ça semblait évident, autant là : Float = Décimal ?! Je vois mal la traduction.



Retenez qu'en réalité il ne s'agit pas de décimaux mais de nombres dits à « virgule flottante ». Nous n'entrerons pas dans le détail pour l'instant, mais lorsque l'ordinateur enregistre un nombre en « virgule flottante », il enregistre le signe, un nombre entier (les chiffres de notre nombre) plus un autre nombre qui indiquera l'emplacement de la virgule. Plus de détails seront donnés dans la quatrième partie.

Valeurs possibles

Si X est une variable de type float, elle peut prendre les valeurs suivantes :

```
0.0 ; 1.0 ; 2.0... 3,40282E38 (un nombre avec 39 chiffres)
-1.0 ; -2.0... -3,40282E38 (un nombre avec 39 chiffres)
1.5 ; - 2.07 ; 3.141592...
```



Vous remarquerez deux choses : tout d'abord 1,2,3... sont notés (et doivent être notés) 1.0, 2.0, 3.0... Ensuite, le nombre Pi n'existe pas même s'il existe le nombre 3.141592 (une valeur approchée). Le type float ne contient donc pas les réels mais des décimaux (indication pour les mathéux). ☺)

Exemple

Voilà ce que donnerait le début d'un programme demandant votre taille et votre poids :

Code : Ada

```
Procedure TaillePoids is
    Taille : Float := 1.85 ;
    Poids : Float := 63.0 ;
begin
...
```

Ou bien, sans affecter de valeur et en déclarant les deux variables en même temps :

Code : Ada

```
Procedure TaillePoids is
    Taille, Poids : Float ;
begin
...
```

Et si vous souhaitez afficher des variables de type float, il faudra ajouter Ada.Float_Text_IO dans la liste des packages. Une instruction Put() spécifique aux float y est enregistrée. Prenez l'exemple suivant :

Code : Ada

```
WITH Ada.Text_IO ;           USE Ada.Text_IO ;
WITH Ada.Float_Text_IO ;    USE Ada.Float_Text_IO ;

Procedure TaillePoids is
    Taille : Float := 1.85 ;
```

```

Poids : Float := 63.0 ;
begin
  Put("Votre taille est de ") ; --On affiche du texte (package
Ada.Text_IO) ;
  Put(Taille) ; --On affiche un Float (package
Ada.Float_Text_IO)
  New line ;
  Put("Votre poids est de ") ; --On affiche du texte (package
Ada.Text_IO)
  Put(Poids) ; --On affiche un Float (package
Ada.Float_Text_IO)
  New line ;
End TaillePoids ;

```

Ce programme est censé afficher une taille (1m85) et un poids (63kg). Mais contrairement à toute attente, voici le résultat :

Code : Console

```
Votre taille est de 1.85000E+00
Votre poids est de 6.30000E+01
```



Qu'est-ce que c'est que ce charabia ? C'est à moi de remettre les chiffres dans l'ordre ?

En fait, les nombres à virgule flottante sont enregistrés dans l'ordinateur en écriture scientifique, c'est à dire sous la forme $1,85 = 1,85 \times 10^0$ et $63 = 6,3 \times 10^1$ (plus de détail seront donnés lors du troisième chapitre sur les variables). Le E majuscule remplace ici la multiplication par une puissance de 10. Je me doute que vous préfériez vous passer de cette écriture scientifique ainsi que de tous les 0 inutiles après la virgule. Pour cela, il suffit d'utiliser deux paramètres, Exp et Aft. Exp est le diminutif d'exposant (ou puissance si vous préférez) ; il suffit d'indiquer 0 pour que celui-ci ne soit pas affiché. Aft est l'abréviation du mot anglais « After », qui signifie « Après » ; celui-ci permet de préciser le nombre minimum de chiffres désirés après la virgule. Si vous indiquez 0, Ada n'affichera de chiffres après la virgule que si nécessaire. Modifions donc le code précédent et observons le résultat :

Code : Ada

```

WITH Ada.Text_IO ;           USE Ada.Text_IO ;
WITH Ada.Float_Text_IO ;     USE Ada.Float_Text_IO ;

Procedure TaillePoids Is
  Taille : Float := 1.85 ;
  Poids : Float := 63.0 ;
begin
  Put("Votre taille est de ") ;
  Put(Taille, Exp => 0, Aft => 0) ;
  New line ;
  Put("Votre poids est de ") ;
  Put(Poids, Exp => 0, Aft => 0) ;
  New line ;
End TaillePoids ;

```

Code : Console

```
Votre taille est de 1.85
Votre poids est de 63
```



Un paramètre `Fore` existe également pour `Ada.Float_Text_IO.Put()`, l'instruction `Put()` pour les nombres à virgule flottante. C'est le diminutif du mot anglais « Before » qui signifie « Avant ». Ce paramètre indique la place nécessaire pour afficher les chiffres avant la virgule. Un équivalent du paramètre `Width` pour les entiers, en quelques sortes.

Le type Character

Définition

Le type `Character` est réservé pour les caractères, c'est-à-dire les lettres. Toutefois, les espaces et les tabulations constituent aussi des caractères de même que le retour à la ligne, la fin de fichier... qui sont des caractères non imprimables. Une variable de type `Character` ne peut pas contenir une phrase mais **un seul** caractère.

Valeurs possibles

Si C est une variable du type `Character`, elle peut prendre les valeurs suivantes : 'a', 'b', 'c', 'z', 'A', 'B', 'C', 'Z', '#', ' ', '%'...



Les lettres doivent être entourées par deux apostrophes, l'espace y compris. De plus, 'a' et 'A' sont deux types Character différents.



Les lettres accentuées (é, è, ñ...) ne sont pas prises en charge. Ce sont les symboles anglo-saxons qui prennent par défaut !

En réalité, les characters ont les valeurs suivantes : 0, 1, 2... 255. Les ordinateurs ne gèrent pas les lettres, mais les chiffres oui ! À chacun de ces nombres est associé un caractère. Dans certains langages, il est donc possible d'écrire '`a`'¹ pour obtenir '`b`'. Mais le langage Ada, qui a un type fort (certains diront excessif), empêche ce genre d'écriture : une lettre plus un nombre, c'est une aberration en Ada. (2) Ne vous inquiétez pas, nous verrons plus tard les opérations possibles sur les `Character`.

Exemple

Si nous voulons pouvoir utiliser les `Character`, il suffit d'avoir écrit `Ada.Text_IO` dans la liste des packages :

Code : Ada

```
With Ada.Text_IO ;
Use Ada.Text_IO ;
```

Et voilà pour le début d'un programme qui demanderait votre initiale :

Code : Ada

```

Procedure Initiales Is
  Ini : Character := 'K' ; --Comme Kajig :-
begin
  ...

```

Affectation

Maintenant que l'on sait déclarer les principaux types de variables, voyons comment leur attribuer une valeur (on dit qu'on leur affecte une valeur). En effet, nous avons réquisitionné une zone de la mémoire de notre ordinateur mais il n'y a encore rien d'écrit dedans. Pour cela, revenons à notre programme `VotreAge` :

Code : Ada

```

with Ada.Text_IO, Ada.Integer_Text_IO ;
use Ada.Text_IO, Ada.Integer_Text_IO ;

Procedure VotreAge Is
  Age : integer ;
Begin
  ...

```

Il y a deux personnes qui peuvent affecter une valeur à la variable `Age` : vous (le programmeur) ou l'utilisateur (peut-être vous aussi, mais pas dans le même rôle).

Affectation par le programmeur (ou le programme)

On va utiliser le symbole « := » vu auparavant, mais cette fois, après le **Begin**.

Code : Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO ;
use Ada.Text_IO, Ada.Integer_Text_IO ;

Procedure VotreAge is
    Age : integer; --On déclare notre variable Age, mais elle
    n'a aucune valeur
Begin
    Age := 27; --On affecte une valeur à la variable Age
End VotreAge ;
```

Ce programme se contentera d'affecter 27 à une variable Age.

Affectation par l'utilisateur

Nous savons comment poser une question à l'utilisateur avec l'instruction Put(). Pour récupérer sa réponse, on utilisera l'instruction Get().

Code : Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO ;
use Ada.Text_IO, Ada.Integer_Text_IO ;

Procedure VotreAge is
    Age : integer;
Begin
    Put("Quel age avez-vous ?"); -- on est poli, on demande l'âge.
    Get(Age); -- Puis, on saisit la réponse de l'utilisateur
    Skip_line; -- Put("Ah, vous avez "); Put(Age); -- on fait une phrase affichant la réponse
End VotreAge ;
```



C'est pas trop compliqué. Sauf que, c'est quoi ce `Skip_line` ?

L'instruction `Skip_line`

`Skip_line` est une instruction que je vous conseille fortement d'écrire après chaque utilisation de l'instruction Get(). Pour comprendre son utilité, reprenons l'exemple du programme TaillePoids :

Code : Ada

```
with Ada.Text_IO, Ada.Float_Text_IO ;
use Ada.Text_IO, Ada.Float_Text_IO ;

Procedure TaillePoids is
    Taille, Poids : float;
Begin
    Put("Quelle est votre taille ?");
    Get(Taille);
    Put("Vous mesurez ");
    Put(Taille);
    Put(" m.");
    Put("Quel est votre poids ?");
    Get(Poids);
    Put("Vous pesez ");
    Put(Poids);
    Put(" kg.");
End TaillePoids ;
```

Puis voyons ce qu'il se passe dans la console.

Code : Console

```
Quelle est votre taille ? 1.8R
Vous mesurez 1.8 m.
Quel est votre poids ?
raised ADA.IO_EXCEPTIONS.DATA_ERROR : a-tiinio.adb:89 instantiated at a-
intio.ads:18
```

Notre utilisateur a tapé 1.8R au lieu de 1.85 (erreur de frappe). Logiquement, le programme plante. Mais regardons en détail ce qu'il s'est passé. La taille enregistrée est 1.8, le R n'a pas été pris en compte. Alors pourquoi le programme ne nous laisse-t-il pas entrer une nouvelle taille ? Pourquoi détecte-t-il une erreur alors que nous n'avons entré aucun poids ? Et qui plus est : pourquoi ne la détecte-t-il pas plus tôt ?

En fait, en tapant 1.8R nous avons envoyé dans le buffer (la mémoire tampon) 1.8 puis R puis le « Entrée » (qui constitue également un caractère). L'instruction Get(Taille) s'est emparée de 1.8, mais le R est resté en mémoire tampon puisqu'à l'évidence, il ne faisait pas partie du nombre demandé. Par conséquent, l'instruction Get(Poids) a regardé dans le buffer ce qu'il y avait à récupérer : la lettre R que vous aviez tapé par erreur ! L'instruction Get(Poids) s'est donc emparée du R qui est un caractère et non un Float. D'où l'erreur.



Et `Skip_line` dans tout ça ? ☺

Et bien l'instruction `Skip_line` aurait évité cette erreur, car Get(Taille) aurait saisi 1.8 pour la taille (comme tout à l'heure) et `Skip_line` aurait ensuite vidé la mémoire tampon : plus de R ou de touche "entrée" en mémoire. Nous aurions ainsi pu saisir notre poids tranquillement.



Attention ! `Skip_line` n'est pas fait pour gérer des erreurs. Il vide seulement la mémoire tampon (appelée buffer), mais pensez à l'écrire après chaque instruction `Get()` !

Voici le code, corrigé avec `Skip_line` :

Code : Ada

```
with Ada.Text_IO, Ada.Float_Text_IO ;
use Ada.Text_IO, Ada.Float_Text_IO ;

Procedure TaillePoids is
    Taille, Poids : float;
Begin
    Put("Quelle est votre taille ?");
    Get(Taille);
    Skip_line;
    Put("Vous mesurez ");
    Put(Taille);
    Put(" m.");
    Put("Quel est votre poids ?");
    Get(Poids);
    Skip_line;
    Put("Vous pesez ");
    Put(Poids);
    Put(" kg.");
End TaillePoids ;
```

Compléments

Cette dernière partie est plus technique. Si vous voulez éviter les confusions, je vous conseille de relire ce qui précède et de faire

des exemples (c'est en forgeant que l'on devient forgeron 😊). Toutefois, je vous invite à revenir lire ce qui suit, aussitôt que vous maîtriserez les notions de types car les constantes et les attributs (surtout les attributs) nous seront fort utiles par la suite.

Constantes

Une variable peut... varier. Étonnant non ? 😊 Ce n'est pas parce que vous l'avez initialisée à 15 qu'elle ne peut pas être modifiée plus tard pour valoir, par exemple, 18, bien au contraire. C'est d'ailleurs là tout leur intérêt.

Code : Ada

```
N := 10 ;      --la variable N prend la valeur 10
N:= 13 ;      --Puis la valeur 13
N:= 102 ;     --Et enfin la valeur 102 !
```

Mais il est parfois intéressant de fixer leur valeur une fois pour toute. La variable ne varie alors plus, elle devient une **constante** et doit être déclarée de la manière suivante :

Code : Ada

```
MaVariable : Constant Integer := 15 ;
```

Nous avons ainsi une variable MaVariable qui vaudra toujours 15. Tentez seulement de modifier cette valeur et vous entraînerez un plantage de votre programme (ou plutôt, votre compilateur refusera de compiler votre code).



Mais à quoi servent les constantes ?

Quelques exemples simples. Une variable peut contenir le titre de votre nouveau jeu ou les dialogues des personnages. Mieux vaudrait que le compilateur vous prévienne si ces variables sont malencontreusement modifiées. Autre exemple :

Code : Ada

```
PI : Constant Float := 3.1415926535 ;
```

Chacun comprend que si la valeur de ma variable PI est modifiée, tous mes calculs d'aire de disque, de périmètre de cercle ou de volume de boule... seront faux. Done PAS TOUCHE À MA CONSTANTE !!! 😊

Attributs

Comment obtenir le plus petit et le plus grand Integer ? Comment savoir quel est le 93ème caractère ? Grâce aux attributs, bien sûr. Ce sont des sortes d'instructions qui s'appliquent aux types.

Avant d'en voir quelques uns, voici un petit rappel de vos cours d'anglais : Repeat after me, «It is Susan's dog» signifie «le chien de Susan». Ainsi, le petit mot " * " exprime l'idée d'appartenance et il faut penser à inverser l'ordre des noms par rapport au français. Pourquoi ce rappel ? Eh bien parce que les attributs s'utilisent de la même manière :

Code : Ada

```
N := integer'First ;      --N sera le premier des integer
M := integer'Last ;       --M sera le dernier des integer
C := character'Val(93) ;   --C sera la 93ème valeur des character
P := character'Pos('f') ;  --
P aura pour valeur la position du caractère 'f'
```



N, M et P doivent être déclarés comme des Integer, alors que C est un character !

Nous reverrons plus abondamment les attributs dans les chapitres suivants. Commencez déjà par vous faire la main sur ces quatre là : quel est le plus petit float ? Le plus grand ? Quelle est la position de 'a' et du 'A' ?

Bloc de déclaration

Lorsque vous déclarez une variable ou une constante, cela se fait toujours au sein de la zone réservée aux déclarations : entre les mots clés **IS** et **BEGIN**. Comme nous l'avons dit, c'est à ce moment que votre programme va demander à l'ordinateur s'il peut lui emprunter de la mémoire.



Mais quand notre programme rend-il la mémoire empruntée ?

La mémoire est libérée une fois le programme rendu à l'instruction «**END Bidule** ; » (où Bidule est bien sûr le nom de votre programme). Et jusqu'à ce moment-là, deux variables ne pourront porter le même nom.



Comment je fais si j'ai besoin d'une variable supplémentaire en cours de route ? Et si une variable ne me sert plus à rien au bout de 2 ou 3 lignes ?

La règle de base reste la même : il faut tout déclarer dès le départ et la mémoire réquisitionnée ne sera rendue qu'à la fin du programme. C'est une obligation en Ada et vous verrez, si vous vous essayez à d'autres langages, que c'est une bonne pratique qui structure clairement votre code et évite bon nombre de problèmes. Toutefois, il peut arriver (nous nous en rendrons compte avec les tableaux) que nous ne puissions déclarer une variable dès le départ. Ou bien, une variable peut tout à fait n'avoir qu'un rôle minime à jouer et il est alors intelligent de la détruire une fois inutile afin de dégager de la mémoire. Ada a prévu cette éventualité en proposant des **blocs de déclaration**.

Pour ouvrir un bloc de déclaration, vous devez utiliser le mot clé **DECLARE**. Vous pourrez ensuite déclarer vos variables supplémentaires. Une fois les déclarations terminées, vous indiquerez la reprise d'activité par le mot **BEGIN**. Les variables supplémentaires seront finalement détruites par l'ajout d'un nouveau **END**. Un exemple pour mieux comprendre :

Code : Ada

```
PROCEDURE Mon_Programme IS
BEGIN
  DECLARE
    Ma_Variable_Supplementaire : Integer ;
  BEGIN
    Instructions quelconques faisant intervenir
    Ma_Variable_Supplementaire
  END ; --Fin du bloc de déclaration
END Mon_Programme ;
```

Pour plus de clarté, il est également possible de nommer ce bloc de déclaration de la même façon que l'on déclare une variable :

Code : Ada

```
PROCEDURE Mon_Programme IS
BEGIN
  Bloc Declaration : DECLARE
    Ma_Variable_Supplementaire : Integer ;
  BEGIN
    Instructions quelconques faisant intervenir
    Ma_Variable_Supplementaire
  END Bloc_Declaration ; --Fin du bloc de déclaration
END Mon_Programme ;
```

La variable supplémentaire est ainsi créée à la ligne 5 et pourra être utilisée entre les lignes 6 et 8, mais pas au-delà. Elle sera détruite dès que le programme arrivera à l'instruction «**END Bloc_Declaration** ; ». Il est donc interdit que la variable supplémentaire porte le même nom que les variables de départ, mais il est possible de créer deux variables de même nom si elles sont des blocs de déclaration distincts :

Code : Ada

```
PROCEDURE Mon_Programme IS
BEGIN
    Bloc_Numerol : DECLARE
        X : Float;
    BEGIN
        --Instructions quelconques faisant intervenir X
    END Bloc_Numerol ;
    Bloc_Numer02 : DECLARE
        X : Float;
    BEGIN
        --Instructions quelconques faisant intervenir X
    END Bloc_Numer02 ;
END Mon_Programme ;
```

On comprend sur l'exemple précédent que la première variable X, créée à la ligne 5, meurt à la ligne 8. Il est donc possible d'en créer une nouvelle à la ligne 11 et portant le même nom. Si le bloc de déclaration ne vous sera pas utile pour l'instant (on peut souvent s'en passer en organisant mieux son code), il nous servira tout de même très bientôt.

Nous savons maintenant déclarer une variable et lui affecter une valeur. Nous avons vu également quelques types de variables. Nous serons amenés par la suite à voir d'autres types plus complexes (tableaux, chaînes de caractères, classes, pointeurs...). Avant cela, nous devrons voir un second chapitre sur les variables. C'est en effet intéressant de les avoir enregistrées en mémoire, mais il serait bien plus intéressant de pouvoir les modifier à l'aide de quelques formules mathématiques simples.

En résumé :

- Pour enregistrer vos résultats, vous devez déclarer des variables qui correspondent à des emplacements en mémoire. La déclaration se fait entre les mots clés **IS** et **BEGIN**.
- Toute variable déclarée doit avoir un type bien précis. Nous avons vu 3 grandes familles de types : les entiers (**Integer** ou **Natural**), les flottants qui correspondent aux nombres décimaux (**Float**) et les caractères (**Character**).
- L'assignation d'une valeur à une variable peut se faire de deux façons : soit en laissant le soin à l'utilisateur de choisir soit en codant cette valeur en utilisant l'opération d'affectation (`<:=>`). Les deux méthodes ne sont pas exclusives et vous pouvez bien sûr mixer les deux.
- Si certaines variables ne devraient jamais être modifiées, déclarez-les plutôt comme des constantes avec le terme **CONSTANT**. En cas d'erreur de votre part, le compilateur vous l'indiquera, évitant ainsi le plantage de votre programme.

Variables II : Opérations

Nous allons, dans ce chapitre, aborder les opérations que l'on peut effectuer sur les différents types de variables : Integer, Natural, Float et Character. En effet, il ne suffit pas de créer des variables et de leur affecter une valeur ; encore faut-il qu'elles servent à quelque chose, que l'on effectue quelques opérations avec elles sans quoi, le compilateur GNAT risque de nous renvoyer le message suivant :

Warning : variable «Machine» is never read and never assigned

Autrement dit, votre variable ne sert à rien ! 😊

Un dernier rappel : toutes les opérations d'affectation devront être écrites, je le rappelle, entre **BEGIN** et **END**.

Opérations sur les Integer et les Natural

Voici quelques opérations de base.

Symbol	Opération	Exemple
+	Addition	<code>N := 3 + 4 (résultat 7)</code>
-	Soustraction	<code>N := 5 - 4 (résultat 1)</code>
*	Multiplication	<code>N := 3 * 4 (résultat 12)</code>
/	Division euclidienne (sans nombre à virgule)	<code>N := 7 / 2 (résultat 3 et pas 3.5)</code>
mod	Modulo ("reste" de la division euclidienne)	<code>N := 7 mod 2 (résultat 1)</code>
**	Puissance	<code>N := 5 ** 2 (résultat 25 car 5*5=25)</code>

Je pense que les exemples abordés précédemment parlent d'eux-mêmes, mais ils sont très faciles. Voici quelques exemples un peu plus complexes :

Code : Ada

```
Get(N) ; skip_line ; --on saisit deux variables de type Integer
Get(M) ; skip_line ;

Resultat := N + M + 1 ; --
La variable Resultat prend la valeur résultant
--de l'addition des deux variables N et M
```

Au lieu d'effectuer un calcul avec des valeurs connues, il est possible de faire des calculs avec des variables dont on ne connaît pas, a priori, leur valeur.



Pour que votre programme soit compilé, il faut ABSOLUMENT que Resultat, N et M soient tous des Integer ! Il est toutefois possible de mélanger des Integer et des Natural dans un même calcul car les Natural sont un sous-type des Integer.

Voici une autre utilisation possible de ces opérations :

Code : Ada

```
N := 5 + 5 * 2 ;
M := 15 - (3+2) ;
```

Combien vaudront N et M selon vous ? Si vous me répondez 20 et 14 alors je vais devoir vous rappeler les règles de priorité en Mathématiques. Ada respecte ces priorités. Dans le premier cas, Ada commence par effectuer la multiplication avant d'additionner : N vaut donc 15. Dans le second cas, les parenthèses ont la priorité donc M vaut 10. Un dernier exemple :

Code : Ada

```
Get(Var) ; skip_line ; --on saisit une variable de type integer
Var := Var + 1 ;
```



Comment ? Var est égal à Var+1 ??!

NON ! Le symbole n'est pas = mais := ! Ce n'est pas un symbole d'égalité mais d'affectation ! Supposons que Var soit égal à 5. L'ordinateur commencera par calculer Var + 1 qui vaudra 6. Le code ci-dessus affectera donc 6 à la variable Var, qui valait auparavant 5. On augmente donc la valeur de la variable d'une unité, c'est l'**incrémentation**. Ne vous avais-je pas dit que les variables variaient ? 😊

Opérations sur les float

Opérations élémentaires

Voici quelques opérations sur les float.

Symbol	Opération	Exemple
+	Addition	<code>X := 3.0 + 4.1 (résultat 7.1)</code>
-	Soustraction	<code>X := 5.8 - 4.3 (résultat 1.5)</code>
*	Multiplication	<code>X := 3.5 * 4.0 (résultat 14)</code>
/	Division décimale	<code>X := 7.0 / 2.0 (résultat 3.5)</code>
**	Puissance	<code>X := 36.0 ** 0.5 (résultat 6, la puissance 0.5 équivaut à la racine carré)</code>
ABS	Valeur absolue	<code>X := ABS(-8.7) (résultat 8.7)</code>

Globalement, on retrouve les mêmes opérations qu'avec les Integer et les Natural. Attention toutefois à ne pas additionner, soustraire ou multiplier... un Integer et un Float. Si le symbole de faddition est le même pour les deux types, il s'agit dans les faits de deux instructions distinctes : l'une pour les Integer, l'autre pour les Float.

Ne faites pas d'erreur de casting



Mais comment je fais si j'ai besoin d'ajouter un Float et un Integer ?

C'est très simple, vous devrez effectuer des **conversions** (aussi appelées **cast**). Les conversions de types se font très simplement en Ada : il suffit d'encadrer le nombre ou le calcul à convertir avec des parenthèses et d'écrire le type désiré devant ces mêmes parenthèses. Voici un exemple :

Code : Ada

```
Procedure addition is
  M : integer := 5 ;
  X : float := 4.5 ;
  Res_int : integer ;
  Res_float : float ;
Begin
  Res_int := M + Integer(X) ; Put(Res_int) ;
  Res_float := Float(M) + X ; Put(Res_float) ;
End addition ;
```

L'instruction `Integer(X)` permet d'obtenir la valeur entière de la variable X. Comme X vaut 4.5, `Integer(X)` donnera comme résultat 4. L'instruction `Float(M)` permet d'obtenir l'écriture décimale de la variable M. Comme M vaut 5, `Float(M)` donnera comme résultat 5.0.

Opérations mathématiques



J'aurais besoin de faire des calculs plus poussés, existe-t-il des opérations comme le sinus ou le logarithme ?

Par défaut, ces opérations n'existent pas dans le langage Ada. Mais il existe des packages vous permettant tout de même de les utiliser. Ainsi, si au tout début de votre programme vous ajoutez le package Ada.Numerics, vous aurez alors la possibilité d'utiliser les constantes π et e (e étant la constante de Neper). Exemple :

Code : Ada

```
P := 2 * Pi * 10 ; --Calcule le périmètre d'un cercle de rayon 10
put(e**2) ; --Calcule et affiche e au carré
```

Mais le package Ada.Numerics.Elementary_Functions vous sera plus utile encore car il contient les fonctions trigonométriques et logarithmiques essentielles. Vous trouverez entre autres :

Symbol	Opération	Exemple
Sqrt()	Racine carrée	X := Sqrt(36.0) résulte 6.0 car $6 \times 6 = 36$
Log()	Logarithme népérien	X := Log(e) résulte 1.0
Exp()	Exponentielle	X := Exp(1.0) résultat environ 2.71828 soit la constante de Neper
Cos()	Cosinus	X := cos(pi) résultat -1.0
Sin()	Sinus	X := sin(pi) résultat 0.0
Tan()	Tangente	X := tan(pi) résultat 0.0
ArcCos()	Arc Cosinus	X := arccos(1.0) résultat 0.0
ArcSin()	Arc Sinus	X := arcsin(0.0) résultat 0.0
ArcTan()	Arc Tangente	X := arctan(0.0) résultat 0.0

A ces opérations s'ajoutent également les fonctions hyperboliques : le cosinus hyperbolique (noté cosh), le sinus hyperbolique (noté sinh), la tangente hyperbolique (noté tanh) et leurs réciproques, farcosinus hyperbolique (noté arcosh), l'arcshin hyperbolique (noté arcsinh) et l'arctangente hyperbolique (noté arctanh). Celles-ci s'utilisent de la même manière que leurs homologues trigonométriques.

Opérations sur les caractère

Nous avons déjà parlé dans la partie précédente des attributs. Nous aurons encore besoin d'eux dans cette partie pour effectuer des «opérations» sur les caractère. Voici un tableau récapitulatif de quelques attributs applicables au type character (n'oubliez pas d'écrire l'apostrophe !) :

Attribut	Explication	Exemple
'first'	Renvoie le premier de tous les caractères.	c:=character'first ;
'last'	Renvoie le dernier de tous les caractères.	c:=character'last ;
'pos(#)	Renvoie la position du caractère remplaçant #. Attention , le résultat est un Integer.	n := character'pos('z') ;
'val(#)	Renvoie le #ème caractère. Attention , le symbole # doit être remplacé par un Integer	c:=character'val(165) ;
'succ(#)	Renvoie le caractère suivant	c:=character'succ(c) ;
'pred(#)	Renvoie le caractère précédent	c:=character'pred(c) ;

Nous avons vu au chapitre précédent que certains langages autorisaient l'écriture du calcul «'a' + 1» pour obtenir le caractère 'b'. Pratique et rapide ! Seul souci, on mélange deux types distincts : Character et Integer. Ce genre d'écriture est donc prohibée en Ada. On utilisera donc les attributs de la manière suivante :

Code : Ada

```
character'val(character'pos('a') + 2) ;
```

Expliquer : Ada suit le principe de la priorité des parenthèses donc les calculs seront effectués dans l'ordre suivant :

- `character'pos('a')` va renvoyer le numéro de 'a', sa position dans la liste des caractères disponibles.
- Puis on effectue l'addition (`Position de 'a' + 2`). Ce qui renvoie un nombre entier correspondant à la position de 'c'.
- Enfin, `character'val()` transformera le résultat obtenu en caractère : on devrait obtenir ainsi le caractère 'c' !

Qui plus est, cette opération aurait pu être faite à l'aide de l'attribut `'succ` :

Code : Ada

```
character'succ(character'succ('a')) ;
```

Si ces écritures vous semblent compliquées, rassurez-vous, nous ne les utiliserons pas tout de suite et nous les reverrons plus en détail dans le chapitre sur les fonctions.

Exercices

Pour mettre tout de suite en pratique ce que nous venons de voir, voici quelques exercices d'application.

Exercice 1

Énoncé

Rédigez un programme appelé `Multiple` qui demande à l'utilisateur de saisir un nombre entier et lui retourne son double, son triple et son quintuplé. J'ajoute une difficulté : vous n'aurez le droit qu'à **deux** multiplications. Pas une de plus !

Solution

Secret (cliquez pour afficher)

Code : Ada

```
WITH Ada.Text_IO, Ada.Integer_Text_IO ;
USE Ada.Text_IO, Ada.Integer_Text_IO ;

PROCEDURE Multiple IS
  N : Integer ;
  Double, Triple : Integer ;
BEGIN
  Put("Saisissez un nombre entier : ") ;
  Get(N) ; Skip_Line ;
  Double := 2*N ;
  Triple := 3*N ;
  Put("Le double de ") ; Put(N) ; Put(" est ") ; Put(Double) ; New_Line ;
  Put("Le triple de ") ; Put(N) ; Put(" est ") ; Put(Triple) ; New_Line ;
  Put("Le quintuple de ") ; Put(N) ; Put(" est ") ; Put(Double+Triple) ;
END Multiple ;
```

Exercice 2

Énoncé

Rédigez un programme appelé `Euclide` qui demande deux nombres entiers à l'utilisateur puis renvoie le quotient et le reste de leur division euclidienne (c'est-à-dire la division entière telle que vous l'avez apprise en primaire).

[Solution](#)[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
WITH Ada.Text_IO, Ada.Integer_Text_IO;
USE Ada.Text_IO, Ada.Integer_Text_IO;

PROCEDURE Euclide IS
  N,M : Integer;
BEGIN
  Put("Saisissez le dividende : ");
  Get(N); Skip_Line;
  Put("Saisissez le diviseur : ");
  Get(M); Skip_Line;

  Put("La division de ");
  Put(N);
  Put(" par ");
  Put(M);
  Put(" a pour quotient ");
  Put(N/M);
  Put(" et pour reste ");
  Put(N MOD M);

END Euclide;
```

Exercice 3

[Énoncé](#)

Rédigez un programme appelé Cercle qui demande à l'utilisateur de saisir la longueur d'un rayon d'un cercle et qui affichera le périmètre et l'aire de ce cercle.

Contrainte : le rayon du cercle sera un nombre entier !

 Le périmètre du cercle se calcule en multipliant le rayon par 2 et par Pi.
L'aire du disque se calcule en éllevant le rayon au carré et en le multipliant par Pi.
Pi vaut environ 3,1415926535.

[Solution](#)[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
WITH Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
USE Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;

PROCEDURE Cercle IS
  R : Integer;
  PI : constant float := 3.1415926;
BEGIN
  Put("Saisissez le rayon du cercle : ");
  Get(R); Skip_Line;

  Put("Un cercle de rayon ");
  Put(R);
  Put(" a pour périmètre ");
  Put(2.0*PI*float(R));
  Put(" et pour aire ");
  Put(float(R**2)*Pi);

END Cercle;
```

Exercice 4

[Énoncé](#)

Rédigez un programme appelé Lettres qui demande à l'utilisateur de saisir deux lettres minuscules et qui affichera leur majuscule ainsi que la lettre se trouvant «au milieu» des deux. À défaut, s'il y a deux lettres, le programme choisira la «plus petite».

 Pour obtenir la lettre majuscule, il suffit de lui «soustraire 32» (avec les précautions d'usage bien sûr).

[Solution](#)[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
WITH Ada.Text_IO;
USE Ada.Text_IO;

PROCEDURE Lettres IS
  C1,C2,C3 : character;
BEGIN
  Put("Saisissez une première lettre minuscule : ");
  Get(C1); Skip_Line;
  Put("Sa majuscule est ") ; Put(Character'Val(Character'Pos(C1)-32)) ; New_Line;
  Put("Saisissez une deuxième lettre minuscule : ");
  Get(C2); Skip_Line;
  Put("Sa majuscule est ") ; Put(Character'Val(Character'Pos(C2)-32)) ; New_Line;
  C3 := character'val((character'pos(C1) + character'pos(C2))/2);
  Put("La lettre du milieu est ") ; Put(C3);
END Lettres;
```

Nous voilà désormais armés pour effectuer des calculs, simples ou complexes, sur les différents types de variables connus pour l'heure. Nous aurons l'occasion de les réutiliser par la suite et de comprendre toute leur utilité, notamment pour l'opération MOD, qui doit pour l'instant vous sembler bien inutile (et pourtant elle sera plus utile que vous ne le pensez). Le prochain chapitre devrait nous permettre d'apporter plus de variété encore à nos programmes en proposant des choix.

En résumé :

- Les quatre opérations de base sont accessibles sur le pavé numérique de votre clavier : +, -, * et /.
- Ada respecte les priorités mathématiques et notamment les parenthèses.
- Ada est un langage «à fort type», ce qui implique l'impossibilité de mélanger des types distincts comme additionner un Integer et un Float. Toutefois, des conversions sont possibles.

Les conditions I

L'objectif premier d'un programme informatique est d'automatiser des tâches (souvent répétitives) mais n'oublions pas qu'il va s'adresser à des utilisateurs humains qui auront envie ou besoin d'effectuer des choix que votre programme ne peut anticiper. Nos programmes doivent donc en tenir compte et gérer ces choix : « Si l'utilisateur fait tel choix ALORS faire ceci SINON faire cela ».

Nous profiterons également de ce chapitre pour voir différents opérateurs permettant de comparer nos variables. Le programme étant suffisamment chargé, ce chapitre est scindé en deux parties pour plus de clarté. Dans la seconde partie nous aborderons l'algèbre booléenne, terme effrayant mais qui recouvre des règles de logique qui nous permettront de gérer des conditions plus complexes.

Conditions simples avec IF

Un début en douceur

Nous allons réaliser ici, un programme qui demande à son utilisateur s'il dispose de plusieurs ordinateurs. Cette question n'appellera que deux choix possibles : Oui ou Non. Le programme renverra un message à l'écran qui dépendra de la réponse obtenue. Ce programme peut se décomposer de la manière suivante :

Code : Français

```
AFFICHER la question : "Avez-vous plusieurs ordinateurs ?"
SAISIR la réponse (Oui ou Non)
SI la réponse est oui
    ALORS AFFICHER : "Vous avez bien de la chance."
```

Nous enregistrons la réponse dans une variable `Reponse` qui sera de type Character. L'utilisateur devra taper "`o`", pour oui, et "`n`", pour non. L'instruction `SI` se note `IF` en Ada et l'instruction `ALORS` se note quant à elle `THEN`. D'où le code suivant :

Code : Ada

```
PROCEDURE Questionnaire IS
  Reponse : Character := 'n'; -- on définit Reponse et on lui attribue --par défaut la valeur n (pour non)
BEGIN
  Put ("Avez-vous plusieurs ordinateurs ? (o/n) ");
  Get (Reponse); Skip_line; -- On saisit la réponse et on vide la mémoire tampon
  IF Reponse = 'o' THEN Put ("Vous avez bien de la chance.");
  ELSE Put ("Ha... Dommage.");
  END IF;
END Questionnaire;
```

Mais pourquoi tu as écrit `=` et non pas `:=` comme avant ?

Tout simplement parce que le symbole `=` est utilisé pour affecter une valeur à une variable. Si nous avions écrit `Reponse := 'o'`, alors nous aurions modifier la valeur de `Reponse` en lui attribuant la valeur '`o`'. Or, ici nous posons une question au programme : `Reponse est-elle bien égal à 'o'` ? Le programme se chargera de répondre vrai ou faux. C'est pourquoi, nous utiliserons ici le symbole `=`. D'ailleurs, après une instruction `IF`, le compilateur refuserait que vous utilisez le symbole `:=`.

Autres remarques, l'instruction `IF` doit être clôturée par une instruction `END IF` qui marquera la fin des actions à exécuter. Et s'il ne faut pas de `;` à la fin de l'instruction `IF`, il en faut bien après `END IF` !

Mais ce programme présente de grosses lacunes. Que se passe-t-il si l'utilisateur répond non (`n`) ? Eh bien pour l'instant, c'est très simple : rien.

Une première alternative

Nous allons donc compléter notre code de la manière suivante :

Code : Français

```
AFFICHER la question : "Avez-vous plusieurs ordinateurs ?"
SAISIR la réponse (Oui ou Non)
SI la réponse est oui
  ALORS AFFICHER : "Vous avez bien de la chance."
SINON AFFICHER : "Ha... Dommage."
```

L'instruction `SINON` se traduit par `ELSE` et nous permettra de varier un peu notre programme. Au lieu d'avoir le code suivant :

Code : Ada

```
if Reponse = 'o'
  then Put("Vous avez bien de la chance.");
end if;
```

Nous allons donc écrire :

Code : Ada

```
if Reponse = 'o'
  then Put("Vous avez bien de la chance.");
  else Put ("Ha... Dommage. ");
end if;
```

Désormais, si l'utilisateur répond non (`n`), il recevra tout de même une réponse, différente qui plus est !

Et que se passerait-il si l'utilisateur tapait une autre lettre, comme `z` par exemple ?

Pour avoir la réponse reprenons notre code ! Celui-ci ne teste qu'une seule égalité : la variable `Reponse` est-elle égale à '`o`' ? Donc si `Reponse` vaut '`z`', le programme apportera la même réponse que si elle vaut '`n`', '`a`', '`@`' ou autre ! Dit différemment, nous n'avons pas envisagé tous les cas. L'utilisateur peut tout à fait répondre autre chose que oui ou non. Et là, vous vous demandez s'il n'existerait pas une troisième instruction après `THEN` et `ELSE`. Eh bien non. Pour envisager d'avantage de cas, il va falloir ruser.

Conditions multiples avec IF

L'astuce (pas si compliquée d'ailleurs) est de réaliser plusieurs tests :

Code : Français

```
SI la réponse est oui
  ALORS AFFICHER ("vous avez bien de la chance")
SINON SI la réponse est non
  ALORS AFFICHER ("Ah... Dommage.")
  SINON Afficher ("C'est pas une réponse ça !")
```

Il est en effet possible d'imbriquer plusieurs instructions `IF` les unes dans les autres.

Code : Ada

```
if Reponse = 'o'
  then Put("Vous avez bien de la chance");
  cas où la réponse est oui
    else if Reponse = 'n'
      then Put("Ah... Dommage.");
      --cas où la réponse est non
      else Put("C'est pas une réponse ça !");
      --cas où la réponse est... autre chose
```

```
    end if ;
end if ;
```

Et voilà, le tour est joué ! Allez, plus compliqué maintenant. Si l'utilisateur répond 'P' (pour «p'têt bin que oui, p'têt bin que non») on affichera le message suivant: «Réponses normandes non valides». A vous de jouer!

Secret (cliquez pour afficher)

Code : Ada

```
if Reponse = 'o'
then Put("Vous avez bien de la chance");
else if Reponse = 'n'
then Put("Ah... Dommage..");
else if Reponse = 'p'
then Put("Réponses normandes non valides");
else Put("C'est pas une réponse ça !");
end if ;
end if ;
```

Alors vous avez réussi ? Bien joué ! Regardez toutefois le code que je vous propose. À chaque nouveau **THEN / ELSE**, j'ai ajouté une tabulation (ou 3 espaces avec Adagide). Nous avons déjà vu cette pratique, elle s'appelle **l'indentation**. Elle n'est pas anodine car elle vous permettra de proposer un code aéré, organisé et donc facilement lisible par vous ou un tiers. Prenez cette habitude le plus tôt possible ! C'est un conseil que je ne cesserai de vous répéter.

Toutefois, vous avez dû vous rendre compte que cela devient vite fatigant d'écrire plusieurs **END IF** et d'augmenter de plus en plus le nombre de tabulations pour avoir un code bien indenté. Heureusement les instructions **ELSIF** peuvent être remplacées par une autre : **ELSIF** !

Et ça change quoi ? (À part une lettre en moins)

Et bien, observez par vous-même :

Code : Ada

```
if Reponse = 'o'
then Put("Vous avez bien de la chance");
elsif Reponse = 'n'
then Put("Ah... Dommage..");
elsif Reponse = 'p'
then Put("Réponses normandes non valides");
else Put("C'est pas une réponse ça !");
end if ;
```

Plus besoin de multiplier les **END IF**. Du coup, chaque instruction **ELSIF** se présente comme un «prolongement» du **IF** initial, limitant ainsi l'indentation. Mais il y a une instruction encore plus lisible et particulièrement appropriée pour des choix multiples: la condition multiple avec **CASE**.

Conditions multiples avec CASE

Tester de nombreux cas

Cette nouvelle instruction s'appelle **CASE**. Comme son nom l'indique (si vous êtes un peu anglophone), elle permet de traiter différents CAS (= cases). Alors pour la beauté de l'art, nous allons ajouter un cas supplémentaire. Si l'utilisateur appuie sur 'f' (pour «I don't speak French» = «Je ne parle pas français» pour les anglophones), alors...

Code : Ada

```
case Reponse is --
on indique que l'on va regarder les différents cas possibles pour Reponse
when 'o' => Put("Vous avez bien de la chance.") ; -- si oui
when 'n' => Put("Ah... dommage..") ; -- si non
when 'p' => Put("Réponses normandes non valides");
when 'f' => Put("J'aurais pas du apprendre l'allemand..") ; -- si "not French"
when others => Put("C'est pas une reponse.") ; -- si autre chose
end case ; -- on termine notre instruction comme avec if !
```

Cette nouvelle instruction a des avantages : plus compacte, elle est donc plus claire, plus lisible et surtout plus rapide à taper lorsque les choix s'avèrent nombreux. Elle présente toutefois des limites que nous verrons dans les prochaines parties.

Pour l'instant regardons la structure. La portion de code ci-dessus (on parlera de bloc) est introduite de la même manière que la procédure de votre programme : **PROCEDURE Questionnaire IS**. Elle se termine par **END CASE**. Les flèches sont composées du signe égal (=) suivi du signe «supérieur à» (>). Rappelons enfin que le mot **WHEN** signifie QUAND et que le mot **OTHERS** signifie AUTRE (n'utilisez l'instruction **WHEN OTHERS** qu'après avoir traité tous les cas désirés et afin de traiter tous les autres cas possibles).

Vous avez bien compris ? Alors rien ne vous empêche de trouver d'autres réponses farfelues pour vous exercer. Pensez toutefois à indiquer ces nouvelles réponses possibles à l'utilisateur. Je vous invite à reprendre les parties précédentes si vous avez encore quelques doutes car la partie suivante sera un peu plus compliquée. 😊

En effet, nous pensions avoir fini notre programme, que nous avions vus tous les choix possibles, etc. Et bien non. Que se passera-t-il si l'utilisateur tape 'o' au lieu de 'o', ou bien 'N' au lieu de 'n' ? Pour l'ordinateur, un «o» minuscule et un «N» majuscule sont deux valeurs différentes. Il considérera que c'est une mauvaise réponse et répondra donc «C'est pas une réponse» !

On va devoir créer 4 nouveaux cas pour les majuscules ? 🤔

Non, rassurez-vous. Nous allons toutefois devoir faire un peu de logique : c'est ce qu'on appelle l'algèbre booléenne, mais nous ne l'aborderons pas dans ce chapitre. Si certaines choses ne sont pas claires, je vous conseille de relire ce chapitre car le suivant s'avérera plus compliqué.

Ne rien faire

Une instruction **CASE** doit vérifier toutes les valeurs que peut prendre la variable testée. Dans notre exemple, la variable **Reponse** peut prendre une très grande quantité de valeurs allant de l'abracadabra au symbole dollar. Pour ne pas coder tous ces tests manuellement, Ada a prévu l'instruction **OTHERS** qui regroupe tout ce qui n'a pas été testé.

Mais si je ne veux faire de particulier pour ces autres cas ?

Le langage Ada a tout prévu. Il existe une instruction pour ne rien faire : **NULL**. Il suffira ainsi d'écrire «**WHEN OTHERS => NULL** ;» à la fin pour régler ce menu problème.

L'instruction **NULL** peut jouer d'autres rôles, notamment avec les pointeurs.

Les opérateurs de comparaison et d'appartenance

Les opérateurs de comparaison

Tous nos tests jusqu'à présent étaient destinés à vérifier une égalité. Or, il est possible d'utiliser d'autres symboles mathématiques appelés **opérateurs de comparaison**. En voici un liste.

Opérateur	Signification
=	«est égal à»
/=	«est différent de»
<	«est plus petit que»
>	«est plus grand que»
<=	«est plus petit ou égal à»
>=	«est plus grand ou égal à»

Ainsi, notre programme initial peut-être modifié de la manière suivante :

Code : Français

```
AFFICHER "Combien d'ordinateurs avez-vous?"
SAISIR la réponse
SI la réponse est supérieure ou égal à 2
  ALORS AFFICHER "Génial"
  SINON AFFICHER "C'est toujours ça"
```

D'où le programme suivant :

Code : Ada

```
Procedure Questionnaire2 is
  Reponse : integer := 0 ; --
  on définit la réponse et on l'initialise à 0
begin
  Put("Combien d'ordinateurs avez-vous?");
  Get(Reponse); Skip_line;
  if Reponse >= 2
    then Put("Génial!");
    else Put("C'est toujours ça");
  end if;
end Questionnaire2;
```

À noter que la question posée (Reponse est-elle supérieure ou égale à 2) , aussi appelée **prédictat** peut être remplacée par :

Code : Ada

```
if Reponse>1
  then Put("Génial!");
  else Put("C'est toujours ça");
end if;
```

Ce nouveau prédictat aura la même conclusion. De même, en inversant les instructions suivantes **THEN** et **ELSE** et en écrivant les codes suivants :

Code : Ada

```
if Reponse<=1
  then Put("C'est toujours ça");
  else Put("Génial!");
end if;
```

ou

Code : Ada

```
if Reponse<2
  then Put("C'est toujours ça");
  else Put("Génial!");
end if;
```

... On obtiendra également la même chose.



Règle de logique : le contraire de «<> n'est pas »>> mais bien «<=>». Inversement, le contraire de «<> » est «<=> »!



Ces différents symboles (>, <, >= et <=) ne peuvent être utilisés lors d'une instruction **CASE**. Pensez-y !

L'opérateur d'appartenance

Je dois enfin vous révéler un dernier test possible : le test d'appartenance. Nous avons vu qu'il était interdit de mélanger des variables de type Integer et de type Float, à moins d'effectuer un cast (une conversion). En revanche, je vous ai dit dès le départ que cette interdiction ne valait pas pour les Natural et les Integer, car le type natural n'est finalement qu'un sous-type d'Integer (un produit dérivé en quelque sorte). Voilà qui fait notre affaire : si nous avons une variable N de type Integer et que nous souhaitons tester si elle est positive ou nulle, il suffira de savoir si elle fait partie du sous-type des Natural à l'aide du mot-clé **IN** et de l'attribut '**range**'. Au lieu d'écrire :

Code : Ada

```
IF N >= 0
  THEN ...
```

Il sera possible d'écrire :

Code : Ada

```
IF N IN Natural'range
  THEN ...
```

Petite traduction en Français : « Si N est dans l'intervalle des Natural ». Si N vaut -4, ce prédictat vaudra **FALSE**. De même, il est possible d'effectuer un test pour des intervalles plus restrictifs encore. Si nous souhaitons savoir si N est compris entre 13 et 32, il suffit d'écrire :

Code : Ada

```
IF N IN 13..32
  THEN ...
```



Attention, j'ai bien écrit deux points entre 13 et 32 ! Et pas trois points ! 😊

Pour les utilisateurs de la norme Ada2012

Les expressions IF

Si vous disposez d'un compilateur GNAT récent, celui-ci doit alors répondre à la toute dernière norme du langage, appelée Ada2012. Si ce n'est pas le cas, ce qui suit ne vous concerne pas. Cette norme n'est pas un nouveau langage mais « simplement » une mise à jour, une évolution. Entre autres améliorations, la norme Ada2012 s'est inspirée des langages de programmation dits « fonctionnels » pour proposer le concept d'**expression**. Ces langages (comme F#, Haskell ou OCaml) sont réputés pour donner naissance à des codes très compacts et minimalistes.

L'idée des expressions est de permettre d'utiliser les instructions **IF** au sein même des affectations. Imaginons que suite à la réponse de l'utilisateur vous souhaitez renseigner une autre variable pour aborder votre fichier client. Appelons cette variable **Information** et déclarons-la de type **Integer**. Si l'utilisateur a un ordinateur, **Information** vaudra 1 et 0 sinon. Voici comment nous procéderions avec le langage Ada dans ses 83, 95 ou 2005 :

Code : Ada

```
IF Reponse = 'o'
  THEN Information := 1 ;
  ELSE Information := 0 ;
```

```
END IF ;
```

Et voyez comment la norme Ada2013 nous permet de condenser notre code :

Code : Ada

```
Information := (IF Reponse = 'o' THEN 1 ELSE 0) ;
```

Notez la disparition des points virgules et des deux affectations. L'expression permet d'affecter à notre variable la valeur 1 ou 0 en fonction du résultat du test. Nous pouvons également utiliser cette méthode pour l'affichage :

Code : Ada

```
Put(IF Reponse = 'o' THEN "Vous avez bien de la chance. " ELSE "Ah...  
Dommage.");
```

Et, luxe suprême, il est même possible d'utiliser **ELSIF**, **ELSE IF** ou **THEN IF** ! Mais pour ces deux derniers, il faudra faire usage de parenthèses comme sur l'exemple ci-dessous :

Code : Ada

```
Information := (IF Reponse = 'o' THEN 1 ELSIF Reponse = 'n' THEN 0  
ELSE 2);  
-- OU  
Information := (IF Reponse = 'o' THEN 1 ELSE (IF Reponse = 'n' THEN  
0 ELSE 2));
```

Les expressions CASE

Mais cette importante mise à jour de la norme Ada ne s'arrête pas là. La même possibilité est offerte avec l'instruction **CASE** ! Voici ce que cela donnerait :

Code : Ada

```
Information := (CASE Reponse IS  
WHEN 'o' => 1  
WHEN 'n' => 0  
WHEN OTHERS => 2) ;
```

Il est même possible de combiner **CASE** et **IF** dans une même expression ! Attention toutefois à ne pas vous perdre dans des tests trop compliqués : les expressions sont faites pour condenser le code et pas pour l'alourdir.



Noubliez pas également qu'au final, vos tests doivent renvoyer la valeur à affecter à votre variable.

Toujours plus complexe

Pour terminer, sachez que si votre variable a déjà été initialisée, il est même possible de l'utiliser au sein même de son expression. Regardez par exemple ce code :

Code : Ada

```
Get(x); skip_line;  
x := (if x>= 0 then x else -x) ;
```

L'utilisateur saisit une valeur de x dont on ne connaît pas le signe. La seconde ligne a pour but de la rendre positive : si x est déjà positive ou nulle, on lui affecte sa propre valeur, sinon on lui affecte la valeur -x qui sera nécessairement positive.

Nous avons fait un premier tour des conditions en Ada. L'élément le plus important est la suite d'instruction **IF / THEN / ELSE**, mais n'oubliez pas qu'il existe l'instruction **CASE** pour simplifier votre code. Avec ces quelques instructions en tête, vos programmes devraient d'ores et déjà prendre un peu de profondeur.

En résumé :

- L'instruction **IF** permet de tester une égalité à la fois (ou une égalité). Pour effectuer plusieurs tests, vous devrez imbriquer plusieurs **IF** après des instructions **THEN** ou **ELSE**.
- Pour effectuer un grand nombre d'actions différentes, il vaudra souvent mieux utiliser l'instruction **CASE**. Toutefois, elle ne permettra pas de tester des inégalités.
- Une instruction **CASE** doit proposer une action pour toutes les valeurs possibles de la variable testée. Si vous souhaitez effectuer la même action pour tous les cas « invulnérables », utilisez l'instruction **WHEN OTHERS**.

Les conditions II : les booléens

Nous avons vu, dans le chapitre précédent, l'existence des instructions **IF/ELSIF/CASE**. Mais nous sommes tombés sur un os : nos codes ne prenaient pas en compte les majuscules. Je vous propose donc de repartir du début pour simplifier votre travail de compréhension. Notre code se résumera donc à la version suivante.

Code : Ada

```
if Reponse = 'o'
  then Put("Vous avez bien de la chance. ");
  else Put("Ah... Dommage.");
end if;
```

Voyons maintenant comment gérer les majuscules sans nous répéter.

Introduction aux booléens

Un bref historique

L'algèbre booléenne ou algèbre de Boole (du nom du Mathématicien **George Boole**) est une branche des mathématiques traitant des opérations logiques : on ne manipule plus des nombres mais des propositions qui peuvent être vraies ou fausses. Elle a été très utile en électronique (c'est grâce à elle que nos bétes ordinateurs savent aujourd'hui additionner, multiplier...) et va nous servir aujourd'hui en programmation.



Mais... euh... Je suis nul(e) en Maths.

Pas de problème! Il ne s'agit que de logique donc il n'y aura pas de calcul (au sens habituel du terme). De plus, nous ne ferons qu'effleurer ce pan des Mathématiques.

Qu'est-ce qu'un booléen ?

Tout d'abord, lorsque l'on écrit :

Code : Ada

```
if Reponse = 'o'
```

... il n'y a que deux alternatives possibles : ou cette affirmation est VRAIE (Reponse vaut bien 'o') ou bien elle est FAUSSE (Reponse vaut autre chose). Ces deux alternatives (VRAI/FAUX) sont les deux seules valeurs utilisées en algèbre booléenne. En Ada, elles se notent **TRUE** (VRAI) et **FALSE** (FAUX). Un booléen est donc un objet qui vaut soit VRAI, soit FAUX. En Ada, cela se déclare de la manière suivante :

Code : Ada

```
A : boolean := TRUE;
B : boolean := FALSE;
```

Il est également possible d'affecter à nos variables booléennes le résultat d'un prédictat.

Code : Ada

```
A := (Reponse = 'o')
```



Hein ? La variable A prend la valeur contenue dans Reponse ou bien il prend la valeur 'o' ? Et puis, c'était pas sensé valoir TRUE ou FALSE ?

La variable A prendra bien comme résultat **TRUE** ou **FALSE** et sûrement pas 'o' ! Tout dépend de l'égalité entre parenthèses : si Reponse est bien égale à 'o', alors la variable A vaudra **TRUE**, sinon il vaudra **FALSE**. Il sera ensuite possible d'utiliser notre booléen A dans nos conditions de la façon suivante :

Code : Ada

```
if A
  then...
end if;
```

Cela signifie «Si A est vrai alors...». La variable A remplacera ainsi notre égalité. Par conséquent, il est généralement judicieux de nommer les booléens à l'aide d'adjectifs qualificatifs comme Actif, Present, Majeur ou Est_Majeur ... L'instruction « **IF** Majeur ... » se comprend ainsi plus aisément que « **IF** Majeur = **TRUE** ... »

Les opérateurs booléens

Voici les quelques opérations booléennes que nous allons aborder :

Opérateur	Traduction littérale	Signification
Not	Non	Not A : «il faut que A ne soit pas vrai»
Or	Ou	A or B : «il faut que A ou B soit vrai»
And	Et	A And B : «il faut que A et B soient vrais»
Xor	Ou exclusif	A xor B : «il faut que A ou B soit vrai mais pas les deux »

Ce ne sont pas les seuls opérateurs existants en algèbre booléenne, mais ce sont ceux que vous serez amenés à utiliser en Ada.

La négation avec Not

Revenons à notre programme. Nous voudrions inverser les deux instructions d'affichage de la manière suivante :

Code : Ada

```
if Reponse = 'o'
  then Put("Ah... Dommage.");
  else Put("Vous avez bien de la chance. ");
end if;
```

Chacun comprend que notre programme ne répond plus correctement : si l'utilisateur a plusieurs ordinateurs, le programme lui répond «dommage» ! Nous devons donc changer la condition : le programme doit afficher «Dommage» seulement SI Reponse **vaut pas** 'o' ! La négation d'une instruction booléenne se fait à l'aide de l'instruction **NOT**. D'où le code suivant :

Code : Ada

```
if Not Reponse = 'o' -- si la réponse n'est pas Oui
  then Put("Ah... Dommage."); -- alors on affiche «Dommage»
  else Put("Vous avez bien de la chance. "); -- sinon on affiche le second message
end if;
```

Le programme fait ainsi de nouveau ce qu'on lui demande. Retenez cette astuce car il arrive souvent que l'on se trompe dans la condition ou l'ordre des instructions entre **THEN** et **ELSE**, notamment quand les conditions deviennent plus complexes. De plus, vous pouvez être amenés à ne tester que le cas où une condition ne serait pas remplie, par exemple : « **IF** **NOT** Fichier_Existant ... »

Enfin, nous ne pouvons passer aux opérations suivantes sans faire un peu d'algèbre booléenne. Voici quelques résultats basiques auxquels je vous invite à réfléchir :

- Si A est vrai, alors Non A est faux

- Si A = true alors Not A = false.
- Si A est faux, alors Non A est vrai.
Si A = false alors Not A = true.
- Quelle que soit la valeur du booléen A, Non Non A = A (Not not A = A)

Les opérations Or et And

L'instruction OR

Nous avons vu dans la partie précédente que le code suivant ne gérait pas les majuscules :

Code : Ada

```
if Reponse = 'o'
then Put("Vous avez bien de la chance. ");
else Put("Ah.. Dommage.");
end if ;
```

Il faudrait poser la condition « Si la réponse est o ou O ». C'est-à-dire qu'il suffirait que l'une des deux conditions soit remplie pour que le programme affiche que vous avez de la chance. Pour cela, on utilise l'instruction **OR**. Pour rappel, « OR » signifie « OU » en français.

Code : Ada

```
if Reponse = 'o' or Reponse = 'O'
then Put("Vous avez bien de la chance. ");
else Put("Ah.. Dommage.");
end if ;
```

Encore un peu d'algèbre booléenne : si A et B sont deux propositions (vraies ou fausses, peu importe), l'instruction « A ou B » sera vraie que dans les trois cas suivants :

- si A est vrai
- si B est vrai
- si les deux sont vrais

L'instruction AND

Nouveau jeu ! Inversons, comme dans la partie précédente, les instructions d'affichage. Nous devrions alors écrire une négation :

Code : Ada

```
if not(Reponse = 'o' or Reponse = 'O')
then Put("Ah.. Dommage.");
else Put("Vous avez bien de la chance. ");
end if ;
```

Vous vous souvenez de la partie sur l'instruction **NOT** ? Nous avons fait la même chose : puisque l'on a inversé les instructions d'affichage, nous écrivons une négation dans le prédictif. Mais cette négation peut aussi s'écrire :

Code : Ada

```
if not Reponse = 'o' and not Reponse = 'O'
then Put("Ah.. Dommage.");
else Put("Vous avez bien de la chance. ");
end if ;
```

L'opérateur **AND** signifie ET. Il implique que les deux conditions doivent être remplies **en même temps** : Reponse ne vaut pas 'o' ET Reponse ne vaut pas non plus 'O'. Si une seule des deux conditions n'était pas remplie, l'ensemble serait faux. Pratiquons encore un peu d'algèbre booléenne pour mieux comprendre :

- si A est VRAI et B est FAUX alors «A et B» est FAUX
- si A est VRAI et B est VRAI alors «A et B» est VRAI
- si A est FAUX et B est FAUX alors «A et B» est FAUX

De manière schématique, retenez que la négation d'un OU revient à signifier ET. Le contraire de VRAI OU VRAI est donc FAUX ET FAUX. Bon, je crois qu'il est temps d'arrêter car certains ménages doivent commencer à griller. Je vous invite donc à méditer ce que vous venez de lire.

L'opération XOR (optionnel)

Il nous reste une opération à voir, l'opération **XOR**. Nous ne nous attarderons pas dessus car je préférerais que vous reteniez déjà ce que nous venons de voir sur **NOT**, **OR** et **AND**. Toutefois, si vous voulez en savoir plus, rappelons que la phrase «A ou B» sera vraie dans trois cas :

- si A est vrai
- si B est vrai
- si A et B sont vrais tous les deux

Or, il existe des cas où il ne faut pas que les deux conditions soient vraies en **même temps**. Au lieu d'écrire «(A ou B) et pas (A et B)», il existe l'opération OU EXCLUSIF (**XOR**). Pour que «A XOR B» soit vrai, il faut que :

- A soit vrai
- ou que B soit vrai
- mais pas les deux en **même temps** !

Prenons un exemple. Vous rédigez un programme à destination des négociants (en fruits et légumes ou en matériaux de construction, peu importe). Ceux-ci ont besoin de nouer des contacts avec diverses entreprises, qu'elles soient productrices ou consommatrices de biens. Votre programme a pour but de rechercher de nouveaux clients ou fournisseurs ; il doit donc disposer de deux variables booléennes : **Producteur** et **Consommateur** qui indiquent le statut d'une entreprise. Il dispose également de deux instructions : **Faire_Affaire** et **Laisser_Tomber**. Voici la première idée que nous pourrions avoir :

Code : Ada

```
IF Producteur OR Consommateur
THEN Faire_Affaire ;
ELSE Laisser_Tomber ;
END IF ;
```

On ne contacte une entreprise que si elle est productrice de bien OU consommatrice. Sauf que ce code possède une faille : si l'entreprise a fait le choix de produire elle-même les biens qu'elle consomme, quel intérêt avons-nous à jouer les négociants ? Il faut donc retirer le cas où une entreprise est à la fois productrice et consommatrice. C'est là que **XOR** entre en scène :

Code : Ada

```
IF Producteur XOR Consommateur --on ne retient plus les
entreprises en circuit interne
THEN Faire_Affaire ;
ELSE Laisser_Tomber ;
END IF ;
```

Exercice

Revenons à notre problème initial : maintenant que nous avons vu les opérateurs booléens, vous pourriez reprendre votre programme nommé **Questionnaire** pour qu'il puisse gérer aussi bien les majuscules que les minuscules. Vous allez pouvoir compléter le code du précédent chapitre mais attention, il ne doit pas y avoir de redondance ! Hors de question d'écrire plusieurs fois la même instruction. Vous allez devoir utiliser l'un des opérateurs booléens vus dans ce chapitre. Oui, mais lequel ? À quelle opération cela correspond-il : OU, ET, NON ?

Voici une première solution avec des **ELSIF** :

Secret (cliquez pour afficher)

Code : Ada

```
if Reponse = 'o' or Reponse = 'O'
  then Put("Vous avez bien de la chance.");
elsif Reponse = 'n' or Reponse = 'N'
  then Put("Ah... dommage.");
elsif Reponse = 'f' or Reponse = 'F'
  then Put("Reponses normandes non valides");
elsif Reponse = 'e' or Reponse = 'E'
  then Put("J'aurais pas du apprendre l'allemand...");
else Put("C'est pas une reponse.");
end if;
```

Comme vous avez vu vous en douter, il suffit juste de compléter les blocs **IF/ELSIF** avec une instruction **OR**. Il est évident que votre variable réponse ne peut être en majuscule ET en minuscule en même temps. 😊

Voici une seconde solution avec un **CASE** (un poil plus dur car la solution proposée comporte une astuce) :

Secret (cliquez pour afficher)

Code : Ada

```
case Reponse is
  when 'o' | 'O' => Put("Vous avez bien de la chance.");
  - si oui
  when 'n' | 'N' => Put("Ah... dommage.");
  - si non
  when 'p' | 'P' => Put("Reponses normandes non valides");
  - si peut-être
  when 'e' | 'E' => Put("J'aurais pas du apprendre l'allemand...");
  - si pas français
  when others => Put("C'est pas une reponse.");
  - si autre chose
end case;
```

Durant vos essais, vous avez du remarquer que lors de l'instruction **WHEN** il n'était pas possible de faire appel aux opérateurs booléens. Pour remplacer l'instruction **OR**, il suffit d'utiliser le symbole «» (Alt gr + le chiffre 6). Ce symbole se nomme Pipe (prononcez « Païpe », c'est de l'Anglais). Oui, je sais vous ne pouvez pas le trouver par vous-même. 😊

Priorités booléennes et ordre de test (Supplément)

Cette ultime partie doit être vue comme un complément. Elle est plutôt théorique et s'adresse à ceux qui maîtrisent ce qui précède (ou croient le maîtriser 😊). Si vous débutez, jetez-y tout de même un œil pour avoir une idée du sujet, mais ne vous affolez pas si vous ne comprenez rien ; cela ne vous empêchera pas de suivre la suite de ce cours et vous pourrez revenir dessus plus tard, lorsque vous aurez gagné en expérience.

Priorités avec les booléens

Nous avons déjà évoqué, dans le chapitre sur les variables et les opérations, la question des priorités dans un calcul. Par exemple, dans l'opération $10 + 5 \times 3$ le résultat est...

Euh... 😊 45?

Argh ! 😊 Non ! Le résultat est 25 ! En effet, on dit que la multiplication est prioritaire sur l'addition, ce qui signifie qu'elle doit être faite en premier (sauf indication contraire à l'aide de parenthèses qui, elles, sont prioritaires sur tout le monde). Et bien des règles de priorités existent également en algèbre booléenne. Découvrons-les à travers ces quelques exemples de la vie courante.

«je viendrais si Albert ET Bernard viennent OU si Clara vient»

Cela revient à écrire : « A **AND** B **OR** C ». Mais que doit-on comprendre ? « (A **AND** B) **OR** C » ou alors « A **AND** (B **OR** C) » ? Quelle opération est prioritaire **AND** ou **OR** ? Réfléchissons à notre exemple, si Clara est la seule à venir, tant pis pour Albert et Bernard, je viens.

- Dans le cas « (A **AND** B) **OR** C » : les parenthèses m'obligent à commencer par le **AND**. Or A et B sont faux tous les deux (Albert et Bernard ne viennent pas) donc (A **AND** B) est faux. Nous finissons avec le **OR** : la première partie est fausse mais C'est vrai (Clara vient pour notre plus grand bonheur), donc le résultat est Vrai ! Je viens ! *A priori*, cette écriture correspondrait.
- Dans le cas « A **AND** (B **OR** C) » : les parenthèses m'obligent à commencer par le **OR**. B est faux (Bernard ne vient pas) mais C'est vrai (Clara vient 😊) donc (B **OR** C) est vrai. Nous finissons avec le **AND** : la deuxième partie est vraie et A est faux (Albert ne vient pas), donc le résultat est... Faux ! Je suis sensé ne pas venir ! Apparemment, cette écriture ne correspond pas à ce que l'on était sensé obtenir.

Conclusion ? Il faut commencer par le **AND**. Si nous enlevons les parenthèses, alors **AND** est prioritaire sur **OR**. Le **ET** peut être comparé à la multiplication ; le **OU** peut être comparé à l'addition.

i Pour ceux qui feraient (ou auraient fait) des probabilités au lycée, le **ET** peut-être assimilé à l'intersection **∩** et donc à la multiplication, tandis que le **OU** peut-être assimilé à l'union **U** et donc à l'addition. Gardez ces liens en tête, cela vous facilitera l'apprentissage des formules ou de l'algèbre booléenne.

Eh XOR et NOT ?

XOR a la même priorité que **OR**, ce qui est logique. Pour **NOT**, retenez qu'il ne s'applique qu'à un seul booléen, il est donc prioritaire sur les autres opérations. Si vous voulez qu'il s'applique à toute une expression, il faudra user des parenthèses.

Ordre de test

L'intérêt de **IF**, n'est pas seulement de distinguer différents cas pour appliquer différents traitements. Il est également utile pour éviter des erreurs qui pourraient engendrer un plantage en bonne et due forme de votre programme. Par exemple, prenons une variable **n** de type naturel : elle ne peut pas (et ne doit pas) être négative. Donc avant de faire une soustraction, on s'assure qu'elle est suffisamment grande :

Code : Ada

```
if n >= 5
  then n := n - 5;
end if;
```

Ce que nous allons voir ici a notamment pour intérêt d'éviter des tests qui engendraient des erreurs. Nous reviendrons dessus durant les chapitres sur les tableaux et les pointeurs pour illustrer mon propos. Nous resterons ici dans la théorie. Supposons que nous voulions ordonner nos tests :

Vérifie si A est vrai ET SEULEMENT SI C'EST VRAI, vérifie si B est vrai

Quel intérêt ? Je ne rentrerai pas dans un exemple en programmation, mais prenons un exemple de la vie courante. Que vaut-il mieux faire ?

*Vérifier si ma sœur qui est ceinture noire de judo n'est pas sous la douche ET vérifier si il y a encore de l'eau chaude.
ou
Vérifier si ma sœur qui est ceinture noire de judo est sous la douche ET SEULEMENT SI ELLE N'EST PAS SOUS LA DOUCHE vérifier si il y a encore de l'eau chaude.*

Les vicieux tenteront la première solution et finiront avec une pomme de douche greffée à la place de l'oreille. Les prudents opteront pour la deuxième solution. Autrement dit, en programmation, la première méthode peut occasionner un plantage, pas la seconde. Maintenant voyons comment implémenter cela en Ada :

Code : Ada

```
if A=true AND THEN B=true    --remarque : il n'est pas utile  
d'écrire "true"  
THEN ...  
ELSE ...  
END IF ;
```

Ainsi, on ne testera le prédictat "B=true" que si auparavant A était vrai. Autre possibilité :

Vérifie si A est vrai OU SI CE N'EST PAS VRAI, vérifie si B est vrai

Exemple de la vie courante. Vaut-il mieux...

Vérifier si il y a des steaks hachés dans le frigo ou chez le voisin?

ou

Vérifier si il y a des steaks hachés dans le frigo ou, SI VRAIMENT ON A TOUT FINI HIER aller vérifier chez le voisin

Je ne sais pas vous, mais avant d'aller demander au voisin, je regarde si je n'ai pas ce qu'il faut dans mon frigo. Idem en programmation, si l'on peut économiser du temps-processor et de la mémoire, on ne s'en privé pas. Donc pour implémenter cela, nous écrirons :

Code : Ada

```
if A=true OR ELSE B=true    --remarque : il n'est pas utile d'écrire  
d'après  
THEN ...  
ELSE ...  
END IF ;
```



Petite traduction pour les éternels anglophobes : **AND THEN** = «et alors» / **OR ELSE** = «et sinon»

Les mots réservés **THEN** et **ELSE** peuvent donc, s'ils sont combinés respectivement avec **AND** et **OR**, être utilisés au moment du prédictat. Cela vous montre toute la souplesse du langage Ada cachée sous l'apparence rigide.

L'algèbre booléenne peut vite s'avérer compliquée, alors n'hésitez pas à faire des schémas ou des tableaux si vous commencez à vous perdre dans vos conditions : quand on débute, bon nombre de bogues proviennent de conditions mal ficelées. Malgré cette complexité, retenez l'existence de **NOT/OR/AND**. Ces trois opérateurs vous serviront régulièrement et permettront d'effectuer des tests plus complexes.

En résumé :

- Une variable de type Boolean ne peut prendre que deux valeurs : **TRUE** ou **FALSE**. Ces valeurs peuvent être affectées à votre variable soit directement, soit par un prédictat (dont la valeur est en général inconnue du programmeur).
- L'opérateur **OR** exige qu'au moins l'un des deux prédictats ou booléens soit vrai. L'opérateur **AND** exige que les deux prédictats ou booléens soient vrais. L'opérateur **XOR** exige qu'un seul des deux prédictats ou booléens soit vrai.
- Les négations (**NOT**) sont à manier avec prudence et parcimonie car elles conduisent très vite à des formules booléennes complexes. Ainsi, nous avons vu que « **NOT (A OR B)** » était identique à « **(NOT A) AND (NOT B)** » ou encore que « **(A OR B) AND NOT (A AND B)** » pouvait se simplifier en « **A XOR B** ».
- Rappelez-vous que **AND** est prioritaire sur **OR**. En cas de doute, mieux vaut utiliser trop de parenthèses que de risquer le bogue.

Les boucles

Nous avons terminé le chapitre sur les conditions qui était quelque peu ardu. Maintenant nous allons nous intéresser à un nouveau problème : nos programmes auront souvent besoin de répéter plusieurs fois la même action et il est hors de question de jouer du copier/coller, d'autant plus que le nombre de répétitions ne sera pas nécessairement connu à l'avance ! Notre code doit rester propre et clair.

C'est ce pourquoi ont été créées les boucles. Il s'agit d'une nouvelle instruction qui va répéter autant de fois qu'on le souhaite une même opération. Intéressé ? Pour comprendre tout cela, nous allons créer un programme appelé Ligne qui affichera une ligne de '#' (leur nombre étant choisi par l'utilisateur). La structure de notre programme devrait donc ressembler à cela :

Code : Ada - ligne.adb

```
WITH Ada.Text_IO ;      USE Ada.Text_IO ;
WITH Ada.Integer_Text_IO ; USE Ada.Integer_Text_IO ;

Procedure Ligne Is
    Nb : integer ;
begin
    Put("Combien de dièses voulez-vous afficher ?") ;
    Get(Nb) ; Skip_line ;
    --Bloc d'affichage
end Ligne ;
```

La boucle Loop simple

Principe général

Avec ce que nous avons vu pour l'instant, nous aurions tendance à écrire notre bloc d'affichage ainsi :

Code : Ada

```
case Nb is
    when 0 => Null ; --ne rien faire
    when 1 => Put("#") ;
    when 2 => Put("##") ;
    when 3 => Put("###") ;
    when 4 => Put("####") ;
    when 5 => Put("#####") ;
    when others => Put("#####") ;
end case ;
```

Le souci c'est que l'utilisateur peut très bien demander 21 dièses ou 24 et que nous n'avons pas tellement envie d'écrire tous les cas possibles et imaginables. Nous allons donc prendre la partie de n'afficher les '#' que un par un, et de répéter l'action autant de fois qu'il le faudra. Pour cela, nous allons utiliser l'instruction **LOOP** que nous écrirons dans le bloc d'affichage.

Code : Ada

```
LOOP          --début des instructions qu'il faut répéter
    Put('#') ;
END LOOP ;   --indique la fin des instruction à répéter
```

Ceci définit une **boucle** qui répétera à l'infini tout ce qui est écrit entre **LOOP** et **END LOOP**. Plus exactement, on parle de **boucle itérative**, l'itération étant ici synonyme de répétition. Génial non ?

Arrêter une itération

Et elle s'arrête quand ta boucle ?

Euh... en effet, si vous avez tester ce code, vous avez du vous rendre compte que cela ne s'arrête jamais (ou presque). Ce qui est écrit au-dessus est une **boucle infinie** ! C'est la grosse erreur de débutant. (Bien que les boucles infinies ne soient pas toujours inutiles.) Pour corriger cela, nous allons déclarer une variable appelée **Compteur**.

Code : Ada

```
Compteur : integer := 0 ;
```

Cette variable vaut 0 pour l'instant mais nous l'augmenterons de 1 à chaque fois que nous afficherons un dièse, et ce, jusqu'à ce qu'elle soit égale à la variable **Nb** où est enregistrée le nombre de dièses voulus. On dit que l'on **incrémente** la variable Compteur. Voici donc deux corrections possibles :

Code : Ada

```
loop
    if Compteur = Nb
        then exit ; --
        si on a déjà affiché assez de dièses, alors on "casse" la boucle
    else Put('#') ; --
        sinon on affiche un dièse et on incrémente Compteur
            Compteur := Compteur + 1 ;
        end if ;
    end loop ;
```

Code : Ada

```
loop
    exit when Compteur = Nb ; --
        comme tout à l'heure, on sort si on a affiché assez de dièses
    Put('#') ;
        on peut tranquillement afficher un dièse, car si le nombre
            --d'affichages
        était atteint, la boucle serait déjà cassée
    Compteur := Compteur + 1 ; --et on n'oublie pas d'incrémenter
        notre compteur
end loop ;
```

L'instruction **EXIT** permet de «casser» la boucle et ainsi d'y mettre fin. Dans ce cas, il est important de faire le test de sortie de boucle avant d'afficher et d'incrémenter, afin de permettre de tracer une ligne de zéros dièses. C'est aussi pour cela que Compteur est initialisé à 0 et pas à 1. Si l'on exige l'affichage d'au moins un dièse, alors l'ordre dans la boucle peut être modifié. Toutefois, vous devez réfléchir à l'ordre dans lequel les opérations seront effectuées : il est rarement anodin de décaler l'instruction de sortie d'une boucle de quelques lignes.

Voici un exemple si l'utilisateur tape 3 :

Valeur de Compteur	Test de sortie	Affichage	Incrémantion
0	Négatif	#	+1
1	Négatif	#	+1
2	Négatif	#	+1
3	Positif	STOP	STOP

On affichera bien 3 dièses (pas un de moins ou de plus). Vous voyez alors que si le test de sortie était effectué après l'affichage (ce qui reviendrait à inverser les deux colonnes du milieu), nous obtiendrions l'affichage d'un quatrième #. De même, si la variable **compteur** était initialisée à 1 (ce qui reviendrait à supprimer la ligne **Compteur:=0**), nous n'aurions que deux # affichés. Prenez le temps de réfléchir ou tester les valeurs extrêmes de votre compteur : 0,1, la valeur maximale si elle existe...

Le second code a, ici, ma préférence **IF**, **ELSIF**, **CASE...**. Il n'est d'ailleurs pas impossible d'écrire plusieurs instructions donnant lieu à l'instruction **EXIT**. Une autre variante (plus compacte) serait de ne pas utiliser de variable **Compteur** et de décrémer le variable **Nb** au fur et à mesure (décrémer = soustraire 1 à chaque tour de boucle).

Code : Ada

```
loop
  exit when Nb = 0 ; -- Si Nb vaut 0 on arrête tout
  Nb := Nb - 1 ; -- On n'oublie pas de décrémenter...
  Put('#') ; -- ... et surtout d'afficher notre dièse !
end loop ;
```

Cette dernière méthode est encore plus efficace. N'oubliez pas que lorsque vous créez une variable, elle va monopoliser l'espace mémoire. Si vous pouvez vous passer d'une d'entre elles, c'est autant d'espace mémoire qui sera dégagé pour des tâches plus importantes de votre programme (ou d'un autre). Qui plus est, votre code gagnera également en lisibilité.

Nommer une boucle

Lorsque votre code sera plus complexe et comportera de nombreuses boucles de plusieurs dizaines ou centaines de lignes chacune, il sera sûrement nécessaire de nommer vos boucles pour vous y retrouver. Cela se fait en Ada de la même façon qu'une déclaration : devant le mot clé **LOOP**, il vous suffit d'écrire le nom de votre boucle suivi de deux points. En revanche, il faudra également nommer votre boucle lors du **END LOOP** et éventuellement après l'instruction **EXIT**.

Code : Ada

```
Ma_Boucle : loop
  exit Ma_Boucle when Nb = 0 ; --Il n'est pas obligatoire de
  nommer la boucle ici
  Nb := Nb - 1 ;
  Put('#') ;
end loop Ma_Boucle ; --En revanche il est obligatoire
de la nommer ici
```



Pour ne pas confondre les noms de boucles avec les noms de variables, il est bon d'établir une nomenclature. Par exemple, vous pouvez introduire systématiquement les noms de boucle par un **B_** (**B** Affichage pour « **Boucle d'affichage** ») ou un **L_** (**L** Affichage pour « **LOOP d'affichage** »).

La boucle While

Mais il y a mieux encore. Pour nous éviter d'exécuter un test de sortie au milieu de notre boucle, il existe une variante à notre instruction **LOOP WHILE ... LOOP** ! En Anglais, « **while** » a de nombreux sens, mais dans ce cas de figure il signifie *tant que* ; cette boucle se répétera donc tant que le prédictif inscrit dans les pointillés sera vérifié. Voici un exemple.

Code : Ada

```
while Compteur /= Nb loop
  Compteur := Compteur + 1 ;
  Put('#') ;
end loop ;
```



Contrairement à tout à l'heure, le prédictif est « **Compteur /= Nb** » et plus « **Compteur = Nb** ». **WHILE** indique la condition, non pas de sortie, mais de pérennité de la boucle. C'est généralement une cause de l'échec d'une boucle ou de la création d'une boucle infinie.

Autre possibilité : décrémenter Nb afin de ne pas déclarer de variable Compteur.

Code : Ada

```
while Nb /= 0 loop
  Nb := Nb - 1 ;
  Put('#') ;
end loop ;
```

Enfin, il est possible, comme avec les conditions **IF**, d'utiliser l'algèbre booléenne et les opérateurs **NOT**, **AND**, **OR** (voire **XOR**). Cela évite l'usage d'instructions **EXIT** pour gérer des cas de sortie de boucle supplémentaires. Si vous n'avez plus que de vagues souvenirs de ces opérateurs, révisez le chapitre sur les **booléens**.

La boucle For

Les programmeurs sont des gens plutôt fainéants : ils peuvent limiter les opérations à effectuer, ils les limitent ! Et ça tombe bien : nous aussi ! C'est pourquoi ils ont inventé une autre variante de **LOOP** qui leur permet de ne pas avoir à incrémenter eux-mêmes et de s'épargner la création d'une variable Compteur : la boucle **FOR ... LOOP** ! L'exemple en image :

Code : Ada

```
for i in 1..Nb loop
  put('#') ;
end loop ;
```

Ça c'est du condensé, hein ? Mais que fait cette boucle ? Elle se contente d'afficher des # autant de fois qu'il y a de nombres entre 1 et Nb (les nombres 1 et Nb étant inclus). Notez bien l'écriture **1..Nb** (avec seulement deux points, pas trois), elle indique un **intervalle entier**, c'est à dire une liste de nombres compris entre deux valeurs.



Ce n'est pas une boucle infinie ?

Et non ! Car elle utilise une variable qui lui sert de compteur : **i**. Ce qui présente plusieurs avantages : tout d'abord cette variable **i** n'a pas besoin d'être déclarée, elle est automatiquement créée puis détruite pour cette boucle (et uniquement pour cette boucle, sera inutilisable en dehors). Pas besoin non plus de l'initialiser ou de l'incrémenter (c'est-à-dire écrire une ligne **i := i + 1**) la boucle s'en charge toute seule, contrairement à beaucoup de langage d'ailleurs (comme le C). Pas besoin enfin d'effectuer des tests avec **IF**, **WHEN** ou **CASE** qui alourdissent le code. Je parle parfois pour la boucle **FOR** de boucle-compteur.

Vous vous demandez comment cela fonctionne ? C'est simple. Lorsque l'on écrit « **FOR i IN 1..Nb LOOP** » :

- **FOR** indique le type de boucle
- **i** est le nom de la variable remplaçant Compteur (je n'ai pas réutilisé la variable Compteur car celle-ci nécessitait d'être préalablement déclarée, ce qui n'est pas le cas de **i**, je le rappelle).
- **IN 1..Nb** indique l'**intervalle** dans lequel la variable **i** va varier. Elle commencera avec la valeur 1, puis à la fin de la boucle elle prendra la valeur 2, puis 3... jusqu'à la valeur Nb avec laquelle elle effectuera la dernière boucle. (Pour les mathéux : je mets « **Intervalle** » entre guillemets car il s'agit d'un « **intervalle formé d'entiers seulement** » et non de nombres réels).
- **LOOP** indique enfin le début de la boucle



Tu nous as dit de vérifier les valeurs extrêmes, mais que se passera-t-il si Nb vaut 0 ?

C'est simple, il ne se passera rien. La boucle ne commencera jamais. En effet, « l'intervalle **1..Nb** » est ce que l'on appelle un « **integer'range** » en Ada et les bornes 1 et Nb ne sont pas interchangeables dans un **integer'range** (intervalle fait d'entiers). La borne inférieure est forcément 1, la borne supérieure est forcément Nb. Si Nb vaut 0, alors la boucle **FOR** passera en revue tous les nombres entiers (integer) supérieurs à 1 et inférieurs à 0. Comme il n'en n'existe aucun, le travail sera vite fini et le programme n'entrera jamais dans la boucle.

Quant au cas Nb vaudrait 1, **integer'range 1..1** contient un nombre : 1 ! Donc la boucle **FOR** ne fera qu'un seul tour de piste ! Magique non ?



Et si je voulais faire le décompte « en sens inverse » ? En décrémentant plutôt qu'en incrémentant ?

En effet, comme vous l'avez sûrement compris, la boucle **FOR** parcourt l'intervalle **1..Nb** en incrémentant sa variable **i**, jamais en la décrémentant. Ce qui fait que si Nb est plus petit que 1, alors la boucle n'aura pas lieu. Si vous souhaitez faire « l'inverse », parcourir l'intervalle en décrémentant, c'est possible, il suffira de le spécifier avec l'instruction **REVERSE**. Cela indiquera que vous souhaitez parcourir l'intervalle « à rebours ». Voici un exemple :

Code : Ada

```
for i in reverse 1..nb loop
...
end loop ;
```

Attention, si Nb est plus petit que 1, la boucle ne s'exécutera toujours pas !

Les antiquités : l'instruction `GOTO`

Nous allons, dans cette partie, remonter aux origines des boucles et voir une instruction qui existait bien avant l'apparition des `LOOP`. C'est un héritage d'anciens langages, qui n'est guère utilisé aujourd'hui. C'est donc à une séance d'archéologie (voire de paléontologie) informatique à laquelle je vous invite dans cette partie, en allant à la découverte de l'instruction `GOTO` (ou `gotosaurus`). 😊

Cette instruction `GOTO` est la contraction des deux mots anglais «*go to*» qui signifient «*aller à*». Elle va renvoyer le programme à une balise que le programmeur aura placé en amont dans son code. Cette balise est un mot quelconque que l'on aura écrit entre chevrons <>...>. Par exemple :

Code : Ada

```
<<debut>> --on place une balise appelée "debut"
Put('#') ;
goto debut ; --cela signifie "maintenant tu retournes à <<debut>>"
```

Bien sûr il est préférable d'utiliser une condition pour éviter de faire une boucle infinie :

Code : Ada

```
<<debut>>
if Compteur <= Nb
then Put('#') ;
    Compteur := Compteur + 1 ;
    goto début ;
end if ;
```

Cette méthode est très archaïque. Elle était présente en Basic en 1963 et fut très critiquée car elle générât des codes de piètre qualité. À noter que dans le dernier exemple l'instruction `GOTO` se trouve au beau milieu de notre bloc `IF` ! Difficile de voir rapidement où seront les instructions à répéter, d'autant plus qu'il n'existe pas de «`END GOTO`» : cette instruction ne fonctionne pas par bloc ce qui dégrade la lisibilité du code. Cette méthode est donc à proscrire autant que possible. Préférez-lui une belle boucle `LOOP WHILE` ou `FOR` (quoique `GOTO` m'aît déjà sauvé la mise deux ou trois fois 😊).

Boucles imbriquées

Nous voudrions maintenant créer un programme appelé `Carre` qui afficherait un carré de '#' et non plus seulement une ligne.

Méthode avec une seule boucle (plutôt mathématique)

Une première possibilité est de se casser la tête. Si on veut un carré de côté n, alors cela veut dire que le carré aura n lignes et n colonnes, soit un total de $n \times n = n^2$ dièses.



Sur un exemple : un carré de côté 5 aura 5 lignes comprenant chacune 5 dièses soit un total de $5 \times 5 = 25$ dièses. On dit que 25 est le carré de 5 et on note 5^2 (c'est la multiplication de 5 par lui-même). De même, $3^2 = 9$, $6^2 = 36$, $10^2 = 100$...

Donc nous pourrions utiliser une boucle allant de 1 à Nb^2 (de 1 à 25 pour notre exemple) et qui affichera un retour à la ligne de temps en temps.



Pour un carré de côté 5, il faut aller à la ligne au bout de 5, 10, 15, 20,... dièses soit après chaque multiple de 5. Pour savoir si un nombre X est multiple de 5, nous utiliserons le calcul «`X MOD 5`» qui donne le reste de la division de X par 5. Si X est un multiple de 5, ce calcul doit donner 0.

Il faudra donc une boucle du type «`FOR i IN 1..Nb**2`» plus un test du type «`IF i MOD Nb = 0`» dans la boucle pour savoir si l'on doit aller à la ligne. A vous d'essayer, je vous fournis une solution possible ci-dessous :

Secret (cliquez pour afficher)

Code : Ada

```
with ada.Text_IO, ada.Integer_Text_IO ;
use ada.Text_IO, ada.Integer_Text_IO ;

procedure Carre is
    Nb : integer ;
begin
    Put("Quel est le côté de votre carré ? ") ;
    Get(Nb) ; Skip_line ;

    for i in 1..Nb**2 loop
        Nb**2 est la notation du carré, on peut aussi écrire Nb*Nb
        Put('#') ;
        if i mod Nb = 0 then
            Si i est multiple de Nb = Si on a déjà affiché toute une ligne
            then New_line ; --On retourne à la ligne
        end if ;
    end loop ;
end Carre ;
```

Vous avez :

- compris ? Très bien, vous devez avoir l'esprit scientifique.
- absolument rien compris ? Pas grave car nous allons justement ne pas faire cela ! C'est en effet un peu compliqué et ce n'est pas non plus naturel comme façon de coder ni reproductible.

Méthode avec deux boucles (plus naturelle)

La méthode précédente est quelque peu tirée par les cheveux mais constitue toutefois un bon exercice pour manipuler les opérations puissance (Nb^{**2}) et modulo (i `MOD` Nb). Voici une méthode plus intuitive, elle consiste à imbriquer deux boucles l'une dans l'autre de la manière suivante :

Code : Français

```
Boucle1
  Boucle2
    fin Boucle2
fin Boucle1
```

La Boucle2 sera chargée d'afficher les dièses d'une seule ligne. La Boucle1 sera chargée de faire défiler les lignes. À chaque itération, elle relancera la Boucle2 et donc affichera une nouvelle ligne. D'où le code suivant :

Code : Ada

```
with ada.Text_IO, ada.Integer_Text_IO ;
use ada.Text_IO, ada.Integer_Text_IO ;

procedure Carre is
    Nb : integer ;
begin
    Put("Quel est le côté de votre carré ? ") ;
    Get(Nb) ; Skip_line ;

    for i in 1..Nb loop
        Boucle chargée d'afficher toutes les lignes
        for j in 1..Nb loop
            Boucle chargée d'afficher une ligne
            Put('#') ;
        end loop ;
        New_line ;
    Après avoir affiché une ligne, pensez à retourner à la ligne
    end loop ;
end Carre ;
```

Et voilà le travail ! C'est simple, clair et efficace. Pas besoin d'en écrire des pleines pages ou d'avoir fait Math Sup. Retenez bien cette pratique, car elle nous sera très utile à l'avenir, notamment lorsque nous verrons les tableaux.

Il est possible d'insérer des instructions **EXIT** dans des boucles **WHILE** ou **FOR**, même si elles sont imbriquées. Toutefois, si l'instruction **exit** est placée dans la boucle n°2, elle ne donnera l'ordre de sortir que pour la boucle n°2, pas pour la n°1.

i Si vous avez besoin de sortir des boucles n°1 et n°2 en même temps (voir même d'une boucle n°3), une astuce consiste à nommer la première boucle comme nous l'avons vu au début de ce chapitre, par exemple **B_Numer01**, et à écrire l'instruction **EXIT B_Numer01**.

? Pourquoi utilisez-vous des lettres i,j... et pas des noms de variable plus explicites ?

Cette habitude n'est pas une obligation, libre à vous de donner à vos variables de boucles les noms que vous souhaitez. Voici toutefois trois raisons, assez personnelles, pour lesquelles je conserverai cette habitude tout au long de ce tutoriel :

1. **Raison 1** : la variable i ou j a un temps de vie très court (restreint à la seule boucle **FOR**) elle ne sera pas réutilisée plus tard. Je devrais donc réussir à me souvenir de sa signification jusqu'à **END LOOP**.
2. **Raison 2** : la variable i ou j servira très souvent comme indice d'un tableau. Exemple : tableau(i) est la i-ème valeur du tableau. Nous verrons cela dans un prochain chapitre, mais vous pouvez comprendre qu'il est plus simple d'écrire tableau(i) que tableau(Nom_de_variable_long_et_complique).
3. **Raison 3** : j'ai suivi une formation Mathématiques-Informatique. Et en «Maths» on a pour habitude d'utiliser la variable i lorsque l'on traite d'objets ayant un comportement similaire aux boucles. Et vous avez sûrement remarqué que nous faisons régulièrement appel aux Mathématiques pour programmer. Par exemple, lorsqu'un mathématicien veut additionner les carrés de tous les nombres compris entre 1 et N, il écrit généralement : $\sum_{i=1}^N i^2$.

Exercices

Voilà voilà. Si maintenant vous voulez vous exercer avec des boucles imbriquées, voici quelques idées d'exercices. Je ne propose pas de solution cette fois. À vous de vous creuser les méninges.

Exercice 1

[Secret \(cliquez pour afficher\)](#)

Affichez un carré creux :

Code : Console

```
######
#   #
#   #
#   #
######
```

Exercice 2

[Secret \(cliquez pour afficher\)](#)

Affichez un triangle rectangle :

Code : Console

```
# 
## 
### 
#### 
#####
```

Exercice 3

[Secret \(cliquez pour afficher\)](#)

Affichez un triangle isocèle, équilatéral, un rectangle... ou toute autre figure.

Exercice 4

[Secret \(cliquez pour afficher\)](#)

Rédigez un programme testant si un nombre N est multiple de 2, 3, 4... N-1. Pensez à l'opération mod !

Exercice 5

[Secret \(cliquez pour afficher\)](#)

Rédigez un programme qui demande à l'utilisateur de saisir une somme S, un pourcentage P et un nombre d'années T. Ce programme affichera les intérêts accumulés sur cette somme S avec un taux d'intérêt P ainsi que la somme finale, tous les ans jusqu'à la <t-ième> année.

i Calculer un intérêt à P% sur une somme S revient à effectuer l'opération $S \cdot P / 100$. La somme finale est obtenue en ajoutant les intérêts à la somme initiale ou en effectuant l'opération $S \cdot (1 + P / 100)$.

Exercice 6

[Secret \(cliquez pour afficher\)](#)

Créez un programme qui calcule les termes de la [suite de Fibonacci](#). En voici les premiers termes : 0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8 ; 13 ; 21 ; 34 ; 55... Chaque terme se calcule en additionnant les deux précédents, les deux premiers termes étant 0 et 1.

Maintenant que nous avons acquis de bonnes notions sur les boucles et les conditions, nous allons pouvoir nous intéresser aux sous-programmes. Nous reviendrons également sur les boucles lors du chapitre sur les tableaux. Celui-ci fera appel à tout ce que vous avez déjà vu sur le type des données (`integer`, `float`...), les conditions mais aussi et surtout sur les boucles qui révèleront alors toute leur utilité (notamment la boucle `FOR`). Ce sera également l'occasion de découvrir une autre version de la boucle `FOR` que je ne peux encore vous révéler.

En résumé :

- Lorsqu'il est nécessaire de répéter plusieurs fois des instructions vous devrez faire appel aux boucles.
- Si vous utilisez une boucle simple (`LOOP`), n'oubliez pas d'insérer une condition de sortie avec l'instruction **EXIT**. Vérifiez également que cette condition peut-être remplie afin d'éviter les boucles infinies.
- En règle générale, on utilise plus souvent des boucles **WHILE** ou **FOR** afin que la condition de sortie soit facile à retrouver parmi vos lignes de code. La boucle `LOOP` sera souvent réservée à des conditions de sortie complexes et/ou multiples.
- L'instruction **EXIT** peut être utilisée dans des boucles **WHILE** ou **FOR**, mais limitez cette pratique autant que possible.
- De même qu'il est possible d'imbriquer plusieurs conditions **IF**, vous pouvez imbriquer plusieurs boucles, de même nature ou pas. Cette pratique est très courante et je vous invite à la travailler si vous n'êtes pas à l'aise avec elle.

Procédures et fonctions I

Nous avons vu les variables et leurs différents types et opérations, les conditions, les boucles... autant d'éléments essentiels à l'algorithme et à tout langage, que ce soit l'Ada, le C, le C++, le Python... Nous allons maintenant voir comment organiser et hiérarchiser notre code à l'aide des fonctions et des procédures (présentes dans tous les langages elles aussi). Ces éléments nous seront essentiels pour la suite du cours. Mais, assurez-vous, ce n'est pas bien compliqué (au début en tous cas 😊).

Les procédures

Procédure sans paramètre

Vous avez déjà vu maintes fois le terme procédure. Allons, souvenez-vous :

Code : Ada

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure Main is
begin
...
end Main;
```

Et oui ! Votre programme est constitué en fait d'une grande procédure (plus les packages en en-tête, c'est vrai 😊). Donc vous savez dès lors et déjà comment on la déclare :

Code : Ada

```
Procedure Nom_De_Votre_Procedure is
  --Partie pour déclarer les variables
begin
  --Partie exécutée par le programme
end Nom_De_Votre_Procedure;
```

Quel est donc l'intérêt d'en parler si on sait déjà ce que c'est ? 😊

Eh bien pour l'instant, notre programme ne comporte qu'une seule procédure qui, au fil des modifications, ne cessera de s'allonger. Et il est fort probable que nous allons être amenés à recopier plusieurs fois les mêmes bouts de code. C'est à ce moment qu'il devient intéressant de faire appel à de nouvelles procédures afin de clarifier et de structurer notre code. Pour mieux comprendre, nous allons reprendre l'exemple vu pour les boucles : dessiner un rectangle.

Voici le code de notre programme :

Code : Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Figure is
  nb_lignes : natural;           -- nom de lignes
begin
  Put("Combien de lignes voulez-vous dessiner ?"); -- on demande le nombre de lignes
  get(nb_lignes); skip_line;
  for i in 1..nb_lignes loop
    -- on exécute deux boucles pour afficher notre rectangle
    for j in 1..5 loop
      put("#");
    end loop;
    new line;
  end loop;
end Figure;
```

Comme vous pouvez le voir, ce programme Figure propose d'afficher un rectangle de dièses. Chaque rectangle comporte 5 colonnes et autant de lignes que l'utilisateur le souhaite.

Code : Console

```
Combien de lignes voulez-vous dessiner ? 7
#####
#####
#####
#####
#####
#####
#####
-
```

Nous allons remplacer la deuxième boucle (celle qui affiche les dièses) par une procédure que nous appellerons Affich_Ligne dont voici le code :

Code : Ada

```
Procedure Affich_Ligne is
begin
  for j in 1..5 loop
    put("#");
  end loop;
  new line;
end Affich_Ligne;
```

Euh, c'est bien gentil tout ça, mais je l'écris où ?

C'est très simple. Le code ci-dessus permet de déclarer au compilateur la sous-procédure Affich_Ligne. Puisqu'il s'agit d'une déclaration, il faut écrire ce code dans la partie réservée aux déclarations de notre procédure principale (avec la déclaration de la variable nb_lignes). Voici donc le résultat :

Code : Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Figure is
  Procedure Affich_Ligne is
  begin
    for j in 1..5 loop
      put("#");
    end loop;
    new line;
  end Affich_Ligne;
  nb_lignes : natural;
begin
  Put("Combien de lignes voulez-vous dessiner ?");
  get(nb_lignes); skip_line;
  for i in 1..nb_lignes loop
    Affich_Ligne;
  end loop;
end Figure;
```

Vous remarquerez que dans la partie de la procédure principale contenant les instructions, la deuxième boucle, son contenu et le new_line ont disparu et ont laissé la place à un simple «Affich_Ligne». Cette partie devient ainsi plus lisible et plus compacte.

 Mouais... Bah, écrire tout ça pour gagner 3 lignes dans la procédure principale, moi je dis que c'est pas folichon 

N'en jetez plus !  En effet, cela ne semble pas être une opération très rentable, mais je vous invite à continuer la lecture car vous allez vite comprendre l'intérêt des sous-procédures.

Procédure avec un paramètre (ou argument)

Notre programme est pour l'heure très limité : l'utilisateur ne peut pas choisir le nombre de colonnes ! Nous savons facilement (à l'aide de Put et Get) obtenir un nombre de colonnes mais comment faire pour que notre procédure Affich_Ligne affiche le nombre de dièses désiré ? Nous ne pouvons pas effectuer la saisie de la variable Nb_Colonne dans la procédure Affich_Ligne, car la boucle de la procédure principale demanderait plusieurs fois le nombre de colonnes désiré.

 Ah ! C'est malin ! Avec tes histoires, je me suis cassé la tête et en plus je ne peux plus améliorer mon programme ! 

Attendez ! Il y a une solution ! Nous allons demander le nombre de colonnes en même temps que le nombre de lignes et nous allons modifier notre procédure Affich_Ligne pour qu'elle puisse écrire autant de dièses qu'on le souhaite (pas forcément 5). Voici le nouveau code de notre sous-procédure :

Code : Ada

```
Procedure Affich_Ligne(nb : natural) is
begin
  for j in 1..nb loop
    put('#');
  end loop;
  new line;
end Affich_Ligne;
```

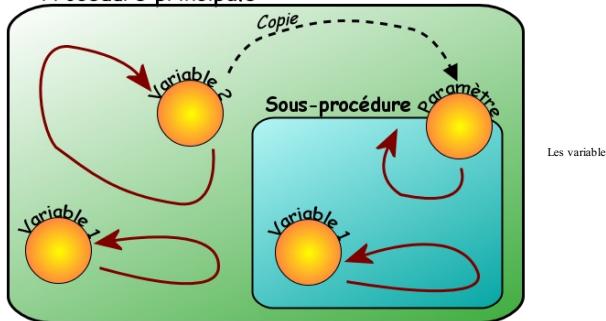
Nous avons remplacé « `j in 1..5` » par « `j in 1..Nb` ». La variable nb correspondra bien évidemment au nombre de dièses désirés. Mais surtout, le nom de notre procédure est maintenant suivi de parenthèses à l'intérieur desquelles on a déclaré la fameuse variable nb.

 J'ai deux questions :

- 1- Pourquoi on la déclare pas dans la zone prévue à cet effet, entre IS et BEGIN ?
- 2- Est-ce normal que la variable nb n'ait jamais été initialisée ?

Cette variable nb est ce que l'on appelle un paramètre. Elle n'est pas déclarée dans la «zone prévue à cet effet», ni initialisée, parce qu'elle vient «de l'extérieur». Petite explication : toute variable déclarée entre IS et BEGIN a une durée de vie qui est liée à la durée de vie de la procédure. Lorsque la procédure commence, la variable est créée ; lorsque la procédure se termine, la variable est détruite : plus moyen d'y accéder ! Cela va même plus loin ! Les variables déclarées entre IS et BEGIN sont inaccessibles en dehors de leur propre procédure ! Elles n'ont aucune existence possible en dehors. Le seul moyen de communication offert est à travers ce paramètre : il permet à la procédure principale de copier et d'envoyer le contenu d'une variable dans la sous-procédure.

Procédure principale



ne sont utilisables que dans la procédure où elles ont été déclarées

Comme l'indique le schéma ci-dessus, il existe deux variables n°1 : une pour la procédure principale et une pour la sous-procédure, mais elles n'ont rien à voir ! Chacune fait sa petite vie dans sa propre procédure. Il serait bien sûr impossible de déclarer dans la même procédure deux variables de même nom. Qui plus est, si une variable n°2 est envoyée comme paramètre dans une sous-procédure, alors elle formera une nouvelle variable, appelée paramètre, distincte de la variable n°2 initiale. La variable n°2 et le paramètre auront la même valeur mais constitueront deux variables distinctes, la seconde n'étant qu'une copie de la première.

Voilà pourquoi la variable nb est déclarée entre parenthèses : c'est un paramètre. Quant à sa valeur, elle sera déterminée dans la procédure principale avant d'être «envoyée» dans la procédure «Affich_Ligne».

Mais revenons à notre procédure principale : Figure. Voilà ce que devient son code :

Code : Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Figure is
  Procedure Affich_Ligne(nb : natural) is
    begin
      for j in 1..nb loop
        put('#');
      end loop;
      new line;
    end Affich_Ligne;
    nb_lignes, nb_colonnes : natural;
  begin
    Put("Combien de lignes voulez-vous dessiner ? ");
    get(nb_lignes); skip_line;
    Put("Combien de colonnes voulez-vous dessiner ? ");
    get(nb_colonnes); skip_line;
    for i in 1..nb_lignes loop
      Affich_Ligne(nb_colonnes);
    end loop;
  end Figure;
```

Nous avons ajouté deux lignes pour saisir le nombre de colonnes voulues. L'instruction «Affich_Ligne» est devenue «Affich_Ligne(nb_colonnes)». Et voilà ! Notre problème initial est résolu ! 

Procédure avec plusieurs paramètres (ou arguments)

 Et bah moi je continue à penser que c'était bien plus simple sans sous-procédure ! Quel intérêt de saucissonner notre code ? 

Toujours pas convaincu ?  Et bien voilà qui devrait mettre fin à vos réticences. Nous allons perfectionner notre programme : il proposera d'afficher soit un rectangle soit un triangle rectangle isocèle (un demi-caré pour les mathophobes ).

Il va donc falloir épurer notre code en créant deux procédures supplémentaires : une pour créer un rectangle (Affich_Rect) et une pour créer le triangle rectangle isocèle (Affich_Tri). Ainsi, notre procédure principale (Figure) se chargera de poser les questions à l'utilisateur, puis elle réorientera vers l'une ou l'autre de ces deux sous-procédures.

La procédure Affich_Rect prendra deux paramètres : le nombre de colonnes et le nombre de lignes.

Code : Ada

```
Procedure Affich_Rect(nb_lignes : natural ; nb_colonnes : natural) is
begin
  for i in 1..nb_lignes loop
    Affich_Ligne(nb_colonnes);
  end loop;
end Affich_Rect;
```



Ici, les deux paramètres sont séparés par un « ; ». Cela permet d'insérer un retour à la ligne au milieu des paramètres pour rendre plus lisible notre code.



Pour fonctionner, Affich_Rect a besoin de la procédure Affich_Ligne, il faut donc que Affich_Ligne soit déclarée avant Affich_Rect !

Et maintenant notre procédure Affich_Tri :

Code : Ada

```
Procedure Affich_Tri(nb : natural) is
begin
  for i in 1..nb loop
    Affich_Ligne(i);
  end loop;
end Affich_Tri;
```

L'utilité des procédures commence à apparaître clairement : il a suffit d'écrire «Affich_Ligne(i)» pour que le programme se charge d'afficher une ligne comportant le nombre de dièses voulus, sans que l'on ait à réécrire notre boucle et à se creuser la tête pour savoir comment la modifier pour qu'elle n'affiche pas toujours le même nombre de dièses. Alors, convaincu cette fois ? Je pense que oui. Pour les récalcitrants, je vous conseille d'ajouter des procédures Affich_Carré, Affich_Tri_Isocèle, Affich_Carré_Vide... et quand vous en aurez assez de recopier la boucle d'affichage des dièses, alors vous adopterez les procédures. 😊

Bien ! Il nous reste encore à modifier notre procédure principale. En voici pour l'instant une partie :

Code : Ada

```
begin
  Put_line("Voulez-vous afficher : "); -- On propose les deux choix : rectangle ou triangle ?
  Put_line("1-Un rectangle ?");
  Put_line("2-Un triangle ?");
  Get(choix); skip_line;

  Put("Combien de lignes voulez-vous dessiner ?"); -- on saisit tout de suite le nombre de lignes
  get(nb_lignes); skip_line;

  if choix=1 then
    selon le choix on procède à l'affichage de la figure
    then Put("Combien de colonnes voulez-vous dessiner ?");
    get(nb_colonnes); skip_line;
    Affich_Rect(nb_lignes, nb_colonnes);
    else Affich_Tri(nb_lignes);
  end if;
end Figure;
```

Notre code est tout de même beaucoup plus lisible que s'il comportait de nombreuses boucles imbriquées !



Mais ? Il y a une erreur ! 🚨 Pour Affich_Rect, les paramètres doivent être séparés par un point virgule et pas par une virgule ! La honte ! 🍀



Et bien non ! Voici une erreur que le compilateur ne nous pardonnerait pas. On utilise le point virgule «;» uniquement lors de la déclaration de la procédure. Lorsque l'on fait appel à la procédure «pour de vrai», on n'utilise que la virgule «,» !

Les fonctions

Qu'est-ce qu'une fonction ? C'est peu ou prou la même chose que les procédures. La différence, c'est que les fonctions renvoient un résultat.

Une bête fonction

Pour mieux comprendre, nous allons créer une fonction qui calculera l'aire de notre rectangle. On rappelle (au cas où certains auraient oublié leur cours de 6ème) que faire du rectangle se calcule en multipliant sa longueur par sa largeur !

Déclaration

Commençons par déclarer notre fonction (au même endroit que l'on déclare variables et procédures) :

Code : Ada

```
function A_Rect(larg : natural ; long : natural) return natural is
  A : natural;
begin
  A:= larg * long;
  return A;
end A_Rect;
```

Globallement, cela ressemble beaucoup à une procédure à part que l'on utilise le mot **FUNCTION** au lieu de **PROCEDURE**. Une nouveauté toutefois : l'instruction **RETURN**. Par **RETURN**, il faut comprendre «résultat». Dans la première ligne, «**RETURN Natural IS** » indique que le résultat de la fonction sera de type natural. En revanche, le «**RETURN A** » de la 5ème ligne a deux rôles :

- Il stoppe le déroulement de la fonction (un peu comme **EXIT** pour une boucle).
- Il renvoie comme résultat la valeur contenue dans la variable **A**.

À noter qu'il est possible de condenser le code ainsi :

Code : Ada

```
function A_Rect(larg : natural ; long : natural) return natural is
begin
  return larg * long;
end A_Rect;
```

Utilisation

Nous allons donc déclarer une variable **Aire** dans notre procédure principale. Puis, nous allons changer la condition de la manière suivante :

Code : Ada

```
...
if choix=1 then
  Put("Combien de colonnes voulez-vous dessiner ?");
  get(nb_colonnes); skip_line;
  Affich_Rect(nb_lignes, nb_colonnes);
  Put("L'aire du rectangle est de "); Put(Aire); put_line(" dièses.");
else Affich_Tri(nb_lignes);
```

```
    end if ;
...

```

Notre variable Aire va ainsi prendre comme valeur le résultat de la fonction A_Rect(). N'oubliez pas le signe d'affectation := ! D'ailleurs, cela ne nous rappelle rien ? Souvenez-vous des attributs !

Code : Ada

```
c := character'val(153) ;
```

Ces attributs agissaient à la manière des fonctions !

Une fonction un peu moins bête (optionnel)



Et faire du triangle ?

C'est un peu plus compliqué. Normalement, pour calculer l'aire d'un triangle rectangle isocèle, on multiplie les côtés de l'angle droit entre eux puis on divise par 2. Donc pour un triangle de côté 5, cela fait $\frac{5 \times 5}{2} = 12,5$ dièses !!! Bizarre. Et puis, si l'on fait un exemple :

Code : Console

```
##
###
####
#####
#####
```

Il y a en fait 15 dièses !! Pour quelle raison ? Et bien, ce que nous avons dessiné n'est pas un «vrai» triangle. On ne peut avoir de demi-dièses, de quart de dièses... La solution est donc un peu plus calculatoire. Remarquez qu'à chaque ligne, le nombre de dièses est incrémenté (augmenté de 1). Donc si notre triangle a pour côté **N**, pour connaître l'aire, il faut effectuer le calcul : $1 + 2 + 3 + \dots + (N - 1) + N$. Les points de suspension remplacent les nombres que je n'ai pas envie d'écrire.

Comment trouver le résultat ? Astuce de mathématicien. Posons le calcul dans les deux sens :

<i>Calcul n°1</i>	1	+	2	+	3	+	...	+	N-2	+	N-1	+	N
<i>Calcul n°2</i>	N	+	N-1	+	N-2	+	...	+	3	+	2	+	1
<i>Somme des termes</i>	N+1	+	N+1	+	N+1	+	...	+	N+1	+	N+1	+	N+1

En additionnant les termes deux à deux, on remarque que l'on trouve toujours **N + 1**. Et comme il y a **N** termes cela nous fait un total de : $N \times (N + 1)$. Mais n'oublions pas que nous avons effectué deux fois notre calcul. Le résultat est donc : $\frac{N \times (N + 1)}{2}$

Pour un triangle de côté 5, cela fait $\frac{5 \times 6}{2} = 15$ dièses ! Bon, il est temps de créer notre fonction A_Triangle !

Code : Ada

```
function A_Triangle(N : natural) return natural is
  A : natural;
begin
  A:= N * (N+1) / 2 ;
  return A ;
end A_Triangle ;
```

Et voilà ce que donne le code général :

Code : Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO ;
use Ada.Text_IO, Ada.Integer_Text_IO ;

procedure Figure is
  Procedure Affich_Ligne(nb : natural) is
  begin
    for j in 1..nb loop
      put('#') ;
    end loop ;
    new_line ;
  end Affich_Ligne ;

  Procedure Affich_Rect(nb_lignes : natural ; nb_colonnes : natural) is
  begin
    for i in 1..nb_lignes loop
      Affich_Ligne(nb_colonnes) ;
    end loop ;
  end Affich_Rect ;

  Procedure Affich_Tri(nb : natural) is
  begin
    for i in 1..nb loop
      Affich_Ligne(i) ;
    end loop ;
  end Affich_Tri ;

  function A_Rect(larg : natural; long : natural) return natural is
    A : natural;
  begin
    A:= larg * long ;
    return A ;
  end A_Rect ;

  function A_Triangle(N : natural) return natural is
    A : natural;
  begin
    A:= N * (N+1) / 2 ;
    return A ;
  end A_Triangle ;

  nb_lignes, nb_colonnes, choix, Aire : natural;

begin
  Put_line("Voulez-vous afficher : ") ;
  Put_line("1-Un rectangle ?") ;
  Put_line("2-Un triangle ?") ;
  Get(choix) ; skip_line;

  Put("Combien de lignes voulez-vous dessiner ? ") ;
  get(nb_lignes) ; skip_line ;

  if choix=1
  then Put("Combien de colonnes voulez-vous dessiner ? ") ;
    get(nb_colonnes) ; skip_line ;
    Affich_Rect(nb_lignes, nb_colonnes) ;
    Aire := A_Rect(nb_lignes, nb_colonnes) ;
    Put("L'aire du rectangle est de ") ; Put(Aire) ; put_line(" dièses.") ;
  else
    Affich_Tri(nb_lignes) ;
    Aire := A_Triangle(nb_lignes) ;
    Put("L'aire du triangle est de ") ; Put(Aire) ; put_line(" dièses.") ;
  end if ;
end Figure ;
```

Bilan

Rendu à ce stade, vous avez du remarquer que le code devient très chargé. Voici donc ce que je vous propose : COMMENTEZ VOTRE CODE ! Exemple :

Code : Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO ;
use Ada.Text_IO, Ada.Integer_Text_IO ;

procedure Figure is
    -----
    --PROCÉDURES--
    -----

    Procedure Affich_Ligne(nb : natural) is
    begin
        for j in 1..nb loop
            put("#");
        end loop;
        new line ;
    end Affich_Ligne ;

    Procedure Affich_Rect(nb_lignes : natural ; nb_colonnes : natural) is
    begin
        for i in 1..nb_lignes loop
            Affich_Ligne(nb_colonnes);
        end loop;
    end Affich_Rect ;

    Procedure Affich_Tri(nb : natural) is
    begin
        for i in 1..nb loop
            Affich_Ligne(i);
        end loop ;
    end Affich_Tri ;

    -----
    --FONCTIONS--
    -----

    function A_Rect(larg : natural; long : natural) return natural is
    begin
        return larg * long ;
    end A_Rect ;

    function A_Triangle(N : natural) return natural is
    begin
        return N * (N+1) / 2 ;
    end A_Triangle ;

    -----
    --VARIABLES--
    -----

    nb_lignes, nb_colonnes,choix : natural;

    -----
    --PROCEDURE PRINCIPALE--
    -----

begin
    Put_line("Voulez-vous afficher : ");
    Put_line("1-Un rectangle ?");
    Put_line("2-Un triangle ?");
    Get(choix); skip_line;
    Put("Combien de lignes voulez-vous dessiner ? ");
    get(nb_lignes) ; skip_line ;

    if choix=1
    then Put("Combien de colonnes voulez-vous dessiner ? ");
        get(nb_colonnes) ; skip_line ;
        Affich_Rect(nb_lignes, nb_colonnes) ;
        Put("L'aire du rectangle est de ") ; Put(A_Rect(nb_lignes, nb_colonne
    else
        Put("L'aire du triangle est de ") ; Put(A_Triangle(nb_lignes)) ; put_
    end if ;
end Figure;
```

J'utilise, pour ma part, de gros commentaires afin de structurer mon code. Nous verrons bien plus tard comment créer des packages afin de libérer notre procédure principale. Autre remarque, dans cet exemple j'ai condensé mon code. J'ai notamment écrit : «`Put(A_Rect(nb_lignes, nb_colonnes))`». Il est en effet possible d'impliquer plusieurs fonctions (ou/et une procédure) les unes dans les autres pour «compacter» votre code. Faites toutefois attention à ce que les types soient compatibles !

Prédefinir ses paramètres

Il est possible de donner une valeur par défaut aux paramètres d'une procédure ou d'une fonction. Par exemple :

Code : Ada

```
Procedure Affich_Ligne(nb : natural := 1) is
```

Ainsi, il sera possible de ne pas spécifier de paramètre. En tapant «`Affich_Ligne()`», le compilateur comprendra tout seul «`Affich_Ligne(1)`». Cette astuce est surtout utile pour des fonctions exigeant de nombreux paramètres dont certains seraient optionnels. Imaginons par exemple une fonction `Affich_Rect` définie comme suit :

Code : Ada

```
procedure Affich_Rect(nb_lignes : natural ;
                      nb_colonnes : natural ;
                      symbole   : character ;
                      plein     : boolean ;
                      affich_a  : boolean ;
                      affich_p  : boolean ) ;
```

On dit qu'il s'agit de la **spécification** de la procédure `Affich_Rect` (dans d'autres langages, comme le C, on parlerait plutôt de **prototype**). On se moque ici de la façon dont la procédure est codée, on se contente d'une simple déclaration. Nous reverrons les spécifications lors du chapitre sur les packages et nous expliquerons alors pourquoi le `IS` a été bizarrement remplacé par un point virgule.

Expliquons plutôt les différents paramètres :

- `nb_lignes` et `nb_colonnes` jouent le même rôle que lors des exemples précédents.
- `symbole` est le caractère qu'il faut afficher pour constituer le rectangle. Auparavant, ce caractère était le dièse (#).
- `plein` est un booléen qui indique si le rectangle est creux (plein vaut `FALSE`) ou plein (plein vaut `TRUE`).
- `affich_a` et `affich_p` sont deux booléens. Si `affich_a` vaut `TRUE`, alors le programme affichera l'aire du rectangle. S'il vaut `FALSE`, alors le programme n'affichera rien. Le paramètre `affich_p` joue le même rôle mais avec le périmètre au lieu de faire.

Nous avons ainsi une procédure vraiment complète. Le seul inconvénient c'est que lorsque l'on veut afficher deux simples rectangles formés de dièses, il faut taper :

Code : Ada

```
Affich_Rect(5,8,'#',true,false,false) ;
Affich_Rect(7,3,'#',true,false,false) ;
```

Honnêtement, les deux premiers, on ne changera pas souvent les paramètres. La procédure est complète, mais un peu trop : il faudra retaper souvent la même chose ('#, `TRUE`, `FALSE`, `FALSE`) en prenant garde de ne pas s'emmêler les pinceaux dans l'ordre des variables (très important l'ordre des variables !).



Eh bien moi, c'est certain ! Je ne ferai jamais de procédure trop compliquée !

Allons ! Cette procédure est très bien, il faut juste identifier les paramètres qui seront optionnels et leur donner une valeur par défaut. Il serait donc plus judicieux de déclarer notre procédure ainsi :

Code : Ada

```
procedure Affich_Rect(nb_lignes : natural;
                      nb_colonnes : natural;
                      symbole    : character := '#';
                      plein      : boolean   := true;
                      affich_a   : boolean   := false;
                      affich_p   : boolean   := false);
```

Ainsi, nous pourrons l'appeler plus simplement :

Code : Ada

```
Affich_Rect(5,8); --Un simple rectangle 5 par 8
Affich_Rect(7,3,'#'); --un rectangle 7 par 3 avec des # à la place des @ !
```

Ah oui, c'est effectivement intéressant. Mais si je veux un rectangle 5 par 8 classique en affichant le périmètre, je peux taper ça ?

Code : Ada

```
Affich_Rect(5,8,true);
```

NON ! Surtout pas ! Le troisième paramètre est sensé être un caractère, pas un booléen !

Mais je fais quoi alors ?

C'est vrai que dans ce cas, on a un souci. Il nous faut respecter l'ordre des paramètres mais certains ne sont pas utiles. Heureusement, il y a une solution !

Code : Ada

```
Affich_Rect(5,8,affich_p => true);
```

Il suffit de spécifier le paramètre auquel on attribue une valeur ! Attention, on n'utilise pas le symbole « := » mais la flèche « => ». Ainsi, l'ordre n'a plus d'importance, on pourrait tout aussi bien noter :

Code : Ada

```
Affich_Rect(5,affich_p => true,nb_colonnes => 8);
Affich_Rect(nb_colonnes => 8,affich_p => true, nb_lignes => 5);
```

D'où l'intérêt de bien choisir ses noms de variables (et de paramètres en l'occurrence).

In, Out, In Out

Nous avons vu précédemment que les paramètres d'une fonction ou d'une procédure ne peuvent pas (et ne doivent pas) être modifiés durant le déroulement de la fonction ou de la procédure. Eh bien... c'est faux ! Enfin non ! C'est vrai, mais disons qu'il y a un moyen de contourner cela pour les procédures.

Paramètres de procédure

Mode in

Lorsque l'on écrit :

Code : Ada

```
procedure Affich_Rect(nb_lignes : natural, nb_colonnes : natural) is
```

Nous écrivons, implicitement, ceci :

Code : Ada

```
procedure Affich_Rect(nb_lignes : in natural, nb_colonnes : in
                      natural) is
```

Les instructions IN indiquent que nos paramètres sont uniquement des paramètres d'entrée. Les emplacements-mémoire qui contiennent leur valeur peuvent être lus mais il vous est interdit d'y écrire : ils ne sont accessibles qu'en lecture. Donc impossible d'écrire « nb_lignes := 5 » au cours de cette procédure.

Mode out

En revanche, en écrivant ceci :

Code : Ada

```
procedure Affich_Rect(nb_lignes : out natural, nb_colonnes : out
                      natural) is
```

... vos paramètres deviendront des paramètres de sortie. Il sera possible d'écrire dans les emplacements-mémoire réservés à nb_lignes et nb_colonnes ! Génial non ? Sauf qu'il vous sera impossible de lire la valeur initiale, donc impossible d'effectuer des tests par exemple. Vos paramètres ne sont accessibles qu'en écriture.

Mode in out

Heureusement, il est possible de mélanger les deux :

Code : Ada

```
procedure Affich_Rect(nb_lignes : in out natural, nb_colonnes : in
                      out natural) is
```

Nous avons alors des paramètres d'entrée et de sortie, accessibles en lecture ET en écriture ! Tout nous est permis !

Pourquoi tous les paramètres ne sont-ils pas en mode IN OUT par défaut ?

Réfléchissons à notre procédure Affich_Rect : est-il judicieux que nos paramètres nb_lignes et nb_colonnes soient accessibles en écriture ? Je ne pense pas. Notre procédure n'a pas besoin de les modifier et il ne serait pas bon qu'elle les modifie. Les laisser en mode IN, c'est prendre l'assurance que nos paramètres ne seront pas modifiés (sinon, le rigoureux compilateur GNAT se chargerait de vous rappeler l'ordre). Et réciproquement, des paramètres en mode OUT n'ont pas à être lu.

Un exemple ? Imaginez une procédure dont l'un des paramètres s'appelle Ca_Marche et servirait à indiquer si il y a eu un échec du programme ou pas. Notre procédure n'a pas à lire cette variable, cela n'aurait pas de sens. Elle doit seulement lui affecter une valeur. C'est pourquoi vous devrez prendre quelques secondes de réflexion avant de choisir entre ces différents modes et

notamment pour le mode **IN OUT**.

Paramètres de fonction



Pourquoi ne parles-tu que des procédures ? Qu'en est-il des fonctions ?

Pour les fonctions, tout dépend de votre compilateur. Pour les normes Ada83, Ada95 et Ada2005, seul le mode **IN** est accessible pour les fonctions. Ce qui est logique puisqu'une fonction renvoie déjà un résultat calculé à partir de paramètres. Il n'y a donc pas de raison pour que certains paramètres constituent des résultats.

Mais cela peut parfois se révéler fort restrictif. C'est pourquoi la norme Ada2012 permet, entre autres améliorations, que les fonctions utilisent des paramètres en mode **IN OUT**. Le mode **OUT** n'est pas accessible : comment calculer le résultat si les paramètres ne sont pas lisibles.

Nous savons maintenant comment mieux organiser nos codes en Ada. Cela nous permettra de créer des programmes de plus en plus complexes sans perdre en lisibilité. Nous reviendrons plus tard sur les fonctions et procédures pour aborder une méthode très utile en programmation : la récursivité. Mais cette partie étant bien plus compliquée à comprendre, nous allons pour l'heure laisser reposer ce que nous venons de voir.

Lors du prochain chapitre théorique, nous aborderons un nouveau type : les tableaux ! Nous entrerons alors dans un nouvel univers (et une nouvelle partie) : celui des types composites ! Je conseille donc à ceux qui ne seraient pas encore au point de revoir les chapitres de la seconde partie. Avant tout cela, il est l'heure pour vous de mettre en application ce que nous avons vu. Le chapitre qui suit est un chapitre pratique : un TP !

En résumé :

- Structurez votre code source en le découplant en sous-programmes, voire en sous-sous-programmes.
- Un sous-programme est généralement une **PROCEDURE**. Mais si vous souhaitez calculer un résultat, il sera préférable d'utiliser les **FUNCTION**.
- La plupart des sous-programmes utilisent des paramètres : pensez à réfléchir à leur mode (lecture et/ou écriture) et, si besoin, à leur attribuer une valeur par défaut pour simplifier leur utilisation ultérieure.
- Si un sous-programme A a besoin d'un sous-programme B, alors il est nécessaire que B soit déclaré avant A, sinon le compilateur ne pourra le voir.

[TP] Le craps

Bien, jusque là je vous ai guidé lors des différents chapitres. Je vous ai même fourni les solutions des exercices (qui a dit «pas toujours» ? 😊). Après ces quelques leçons de programmation en Ada, il est temps pour vous de réaliser un projet un peu plus palpant que de dessiner des rectangles avec des dièses (#). Voilà pourquoi je vous propose de vous lancer dans la réalisation d'un jeu de dé :

Le craps !

Il ne s'agira pas d'un exercice comme les précédents, car celui-ci fera appel à de nombreuses connaissances (toutes vos connaissances, pour être franc). Il sera également plus long et devra être bien plus complet que les précédents. Songez seulement à tous les cas qu'il a fallu envisager lorsque nous avons créé notre programme lors du chapitre sur les conditions. En bien sûr il faudra être aussi exhaustif, si ce n'est davantage.

Rassurez-vous, vous ne seriez pas lâchez en pleine nature ; je vais vous guider. Tout d'abord, nous allons établir les règles du jeu et le cahier des charges de notre programme. Puis nous réglerons ensemble certains problèmes techniques que vous ne pouvez résoudre seuls (le problème du hasard notamment). Pour ceux qui auront des difficultés à structurer leur code et leur pensée, je vous proposerai un plan type. Et finalement, je vous proposerai une solution possible et quelques idées pour améliorer notre jeu.

Les règles du craps

Bon, avant de nous lancer tête baissée, il serait bon de définir ce dont on parle. Le craps se joue avec 2 dés et nous ne nous intéressons pas aux faces des dés mais à leur somme ! Si vos dés indiquent un 5 et un 3 alors vous avez obtenu $5+3=8$! Le joueur mise une certaine somme, puis il lance les dés. Au premier lancer, on dit que le point est à *off*.

- Si le joueur obtient 7 (somme la plus probable) ou 11 (obtenu seulement avec 6 et 5), il remporte sa mise. (S'il avait misé 10\$, alors il gagne 10\$)
- S'il obtient des *craps* (2, 3 ou 12) alors il perd sa mise.
- S'il obtient 4, 5, 6, 8, 9 ou 10, alors le point passe à *on* et les règles vont changer.

Une fois que le point est à *on*, le joueur doit parvenir à refaire le nombre obtenu (4, 5, 6, 8, 9 ou 10). Pour cela, il peut lancer le dé autant de fois qu'il le souhaite, mais il ne doit pas obtenir de 7 !

- S'il parvient à refaire le nombre obtenu, le joueur remporte sa mise.
- S'il obtient 7 avant d'avoir refait son nombre, le joueur perd sa mise. On peut alors hurler : **SEVEN OUT !** Hum... désolé.

Ce sont là des règles simplifiées car il est possible de parier sur les issues possibles (le prochain lancer est un 11, le 6 sortira avant le 7, le prochain lancer sera un craps...) permettant de remporter jusqu'à 15 fois votre mise. Mais nous nous tiendrons pour l'instant aux règles citées au-dessus.

Cahier des charges

Un programmeur ne part pas à l'aventure sans avoir pris connaissances des exigences de son client, car le client est roi (et ici, le client c'est moi 😊). Vous devez avoir en tête les différentes contraintes qui vous sont imposées avant de vous lancer :

- Tout d'abord, les mises se feront en \$ et nous n'accepterons qu'un nombre entiers de dollars (pas de 7\$23, on n'est pas à la boulangerie !)
- Le joueur doit pouvoir continuer à jouer tant qu'il n'a pas atteint les 0\$.
- Un joueur ne peut pas avoir -8\$!!! Scores négatifs interdits. Et pour cela, il serait bon que le croupier vérifie que personne ne mise plus qu'il ne peut perdre (autrement dit, pas de mise sans vérification).
- Le joueur ne peut pas quitter le jeu avant que sa mise ait été soit perdue soit gagnée. En revanche, il peut quitter la partie en misant 0\$.
- L'affichage des résultats devra être lisible. Pas de :

Code : Console

+5

mais plutôt :

Code : Console

Vous avez gagné 5\$. Vous avez maintenant 124\$.

- Le programme devra indiquer clairement quel est le point (*off* ou *on*, et dans ce cas, quel est le nombre ?), si vous avez fait un CRAPS ou un SEVEN OUT, si vous avez gagné, quelle somme vous avez gagné/perdu, ce qu'il vous reste, les faces des dés et leur somme... Bref, le joueur ne doit pas être dans l'incertitude.

Exemple :

Code : Console

Vous avez encore 51\$. Combien misez vous ? 50
Le premier dé donne 1 et le second donne 2. Ce qui nous fait 3.
C'est un CRAPS. Vous avez perdu 50\$, il ne vous reste plus que 1\$.

- Votre procédure principale devra être la plus simple possible, pensez pour cela à utiliser des fonctions et des sous-procédures. Vous êtes autorisés pour l'occasion à utiliser des variables en mode **OUT** ou **IN OUT**. 😊
- Votre code devra comporter le moins de tests possibles : chaque fois que vous testez une égalité, vous prenez du temps processeur et de la mémoire, donc soyez économies.

Compris ? Alors il nous reste seulement un petit détail technique à régler : comment simuler un lancer de dé ? (Oui, je sais, ce n'est pas vraiment un petit détail).

Simuler le hasard (ou presque)

Je vous transmets ci-dessous une portion de code qui vous permettra de générer un nombre entre 1 et 6, choisi au hasard (on dit aléatoirement).

Code : Ada

```
WITH Ada.Numerics.Discrete_Random ;
PROCEDURE Craps IS
    SUBTYPE Intervalle IS Integer RANGE 1..6 ;
    PACKAGE Aleatoire IS NEW Ada.Numerics.Discrete_Random(Intervalle) ;
    USE Aleatoire;
    Hasard : Generator;
    ...

```

❌ Comme il est écrit ci-dessus, vous aurez besoin du package `Ada.Numerics.Discrete_Random` mais vous ne devrez pas écrire l'instruction `USE Ada.Numerics.Discrete_Random` !

❓ Bah, pourquoi ?

Malheureusement, je ne peux pas vous expliquer ce code pour l'instant car il fait appel à des notions sur les packages, l'héritage, la généralité... que nous verrons bien plus tard. Vous n'aurez donc qu'à effectuer un copier-coller. En revanche, je peux vous expliquer la toute dernière ligne : pour générer un nombre «aléatoire», l'ordinateur a besoin d'une variable appelée générateur ou gérant. C'est à partir de ce générateur que le programme créera de nouveaux nombres. C'est là le sens de cette variable `Hasard` (de type `generator`).

Mais vous devrez initialiser ce générateur au début de votre procédure principale une et une seule fois ! Pour cela, il vous suffira d'écrire :

Code : Ada

```
BEGIN
    reset(Hasard) ;
    ...

```

Par la suite, pour obtenir un nombre compris entre 1 et 6 aléatoirement, vous n'aurez à écrire :

Code : Ada

```
MaVariable := random(Hasard) ;
```



Et pourquoi pas une fonction qui renverrait directement un nombre entre 2 et 12 ?

Autant, vous avez les même chances d'obtenir 1, 2, 3, 4, 5 ou 6 en lançant un dé (on dit qu'il y **équiprobabilité**) autant vous n'avez pas la même chance de faire 7 et 2 avec deux dés. En effet, pour avoir 7, vous pouvez faire 1/6, 2/5, 3/4, 4/3, 5/2 ou 6/1 tandis que pour avoir 2, vous ne pouvez faire que 1/1 ! Or nous voulons que notre jeu soit aléatoire (ou presque, je ne rentre pas dans les détails mathématiques), donc nous devrons simuler deux dés et en faire la somme.

Un plan de bataille

Comme promis, pour ceux qui ne sauraient pas comment partir ou qui seraient découragés, voici un plan de ce que pourrait être votre programme.

Vous devez réfléchir aux fonctions essentielles : que fait un joueur de CRAPS ? Il joue **tant qu'** il a de l'argent. Il commence par parier une mise (si elle mise 0, c'est qu'il arrête et qu'il part avec son argent) puis il attaque la première manche (la seconde ne commençant que sous certaines conditions, on peut considérer pour l'instant qu'il ne fait que jouer à la première manche). Voilà donc un code sommaire de notre procédure principale :

Code : Français

```
RESET(hasard) ;
TANT QUE argent /= 0
| PARIER
| | SI mise = 0
| | | ALORS STOP
| | FIN DU SI
| | JOUER LA PREMIÈRE MANCHE
FIN DE BOUCLE
```

À vous de le traduire en Ada. Vous devriez avoir identifié une boucle **WHILE**, un **IF** et un **EXIT**. Vous devriez également avoir compris que nous aurons besoin de deux fonctions/procédures **PARIER** et **JOUER LA PREMIÈRE MANCHE** (nom à revoir bien entendu). La procédure/fonction **PARIER** se chargera de demander la mise désirée et de vérifier que le joueur peut effectivement parier une telle somme :

Code : Français

```
TANT QUE mise trop importante
| DEMANDER mise
FIN DE BOUCLE
```

La fonction/procedure **JOUER LA PREMIÈRE MANCHE** se contentera de lancer les dés, de regarder les résultats et d'en tirer les conclusions.

Code : Français

```
LANCER dés
SI les dés valent
| | 1 ou 11 => on gagne
| | 2, 3 ou 12 => on perd
| autre chose => point := on
| | MANCHE2 avec valeur du point
FIN DE SI
```

Il est possible de réaliser une fonction/procedure **LANCER** qui «jittera les dés» et affichera les scores ou bien on pourra écrire le code nécessaire dans la fonction/procedure **JOUER LA PREMIÈRE MANCHE**. Intéressons-nous à la fonction/procedure **JOUER LA DEUXIÈME MANCHE** :

Code : Français

```
TANT QUE point = on
| SI les dés valent
| | | 7 => on perd
| | point => on gagne
| FIN DE SI
FIN DE BOUCLE
```

Voilà décomposé la structure des différentes fonctions/procédures que vous devrez rédiger. Bien sûr, je ne vous indique pas si vous devez choisir entre fonction ou procédure, si vous devez transmettre des paramètres, afficher du texte, choisir entre **IF** et **CASE**... Je ne vais pas non plus tout faire.

Une solution

Je vous fournis ci-dessous le code d'une solution possible à ce TP, ce n'est pas la seule bien sûr :

Code : Ada

```
WITH Ada.Text_IO, Ada.Integer_Text_IO, Ada.Numerics.Discrete_Random ;
USE Ada.Text_IO, Ada.Integer_Text_IO ;

PROCEDURE Craps IS
--PACKAGES NÉCESSAIRES--
SUBTYPE Intervalle IS Integer RANGE 1..6 ;
PACKAGE Aleatoire IS NEW Ada.Numerics.Discrete_Random(Intervalle) ;
USE Aleatoire;

--FONCTIONS ET PROCÉDURES--

PROCEDURE Lancer (
    D1 : OUT Natural;
    D2 : OUT Natural;
    Hasard : Generator) IS
BEGIN
    D1 := Random(Hasard);
    D2 := Random(Hasard);
    Put(" Vous obtenez un") ; Put(D1,2) ;
    Put(" et un") ; Put(D2,2) ;
    put("." Soit un total de ") ; put(D1+D2,4) ; put_line(" !") ;
END Lancer ;

function parier(argent : natural) return natural is
    mise : natural := 0 ;
begin
    Put(" Vous disposez de ") ; Put(argent) ; put_line(" $.") ;
    while mise = 0 loop
        Put(" Combien souhaitez-vous miser ? ") ;
        Get(mise) ; skip_line ;
        if mise > argent
            then Put_line(" La maison ne fait pas crédit, Monsieur.") ; mise := 0 ;
        else exit ;
        end if ;
    end loop ;
    return mise ;
end parier ;

procedure Manche2(Hasard : generator ; Mise : in out natural ; Argent : in out natural; D1,D2 : Natural; --D1 et D2 indiquent les faces du dé 1 et du 2)
begin
    while point>0 loop
        Lancer(D1,D2,Hasard) ;
        if D1 + D2 = Point
            then Put(" Vous empochez ") ; Put(Mise,1) ; put(" $. Ce qui va faire que vous avez ") ; Put(Argent,1) ; Put(" $. Bravo.") ; new_line ;
        Argent := Argent + Mise ; point := 0 ; mise := 0 ;
        Put(Argent,1) ; Put_line(" $. Bravo.") ; new_line ;
    end loop ;
end Manche2;
```

```

elsif D1 + D2 = 7
    then Put(" Seven Out ! Vous perdez ") ; Put(Mise,1) ; put("$."
        Argent := Argent - Mise ; point := 0 ; mise := 0 ;
        Put(Argent,1) ; Put_line("$ Dommage.") ; new_line ;
    end if ;
end loop ;
end Manche2 ;

procedure Manchel(Hasard : generator ; Mise : in out natural ; Argent : in out natural;
D1,D2 : Natural; --D1 et D2 indiquent les faces du dé 1 et du dé 2
Point : Natural := 0; --0 pour off, les autres nombres indiquent des points
begin
    Lancer(D1,D2,Hasard) ;
    CASE D1, D2 IS
        WHEN 7 | 11 => Put(" Vous empochez ") ; Put(Mise,1) ; put("$."
            Argent := Argent + Mise ; Mise := 0 ;
            Put(Argent,1) ; Put_line("$ Bravo.") ; new_line ;
        WHEN 2 | 3 | 12 => Put(" CRAPS ! Vous perdez ") ; Put(Mise,1) ; p
            Argent := Argent - Mise ; Mise := 0 ;
            Put(Argent,1) ; Put_line("$ Dommage.") ; new_line ;
        WHEN OTHERS => Point := D1 + D2 ;
            Put(" Le point est établi à ") ; Put(point,5)
            Manche2(Hasard,Mise,Argent,Point) ;
    END CASE ;
end Manchel ;

--VARIABLES--
-----
Hasard : Generator; --Ce generator nous servira à simuler le hasard
Argent : Natural := 100; --On commence avec 100$
Mise : Natural := 0; --montant misé, on l'initialise à 0 pour débuter

--PROCÉDURE PRINCIPALE--
-----
BEGIN
    Reset(Hasard) ;
    Put_Line(" Bienvenue au CRAPS ! ! !") ; New_Line ;
    while Argent > 0 loop
        Mise := Parier(Argent) ;
        exit when Mise = 0 ;
        Put(" Les jeux sont faits !") ;
        Manchel(Hasard,Mise,Argent) ;
        new_line ;
    END LOOP ;
    Put("Vous repartez avec ") ; Put(Argent,5) ; put("$ !") ;
END Craps ;

```

Voici quelques pistes d'amélioration, certaines n'étant réalisables qu'avec davantage de connaissances. À vous de trouver vos idées, de bidouiller, de tester... c'est comme ça que vous progresserez. N'ayez pas peur de modifier ce que vous avez fait !

Pistes d'amélioration :

- Prendre en compte tous les types de paris. Pour davantage de détails, aller voir les règles sur ce [site](#).
- Prendre en compte un parieur extérieur, un second joueur.
- Enregistrer les meilleurs scores dans un fichier (pour plus tard).
- Proposer un système de «médailles» : celui qui quitte la partie avec moins de 100\$ aura la médaille de «rookie», plus de 1000\$ la médaille du «Dieu des dés»... à vous d'inventer différents sobriquets.
- Et que pourrait faire le joueur de tout cet argent gagné ? S'acheter une bière, une moto, un cigare ? On peut imaginer un système de récompense.

Les chapitres et notions abordés dans cette partie sont essentiels et seront réutilisés constamment. Il est donc important que vous les ayez assimilés avant de commencer la prochaine partie. N'hésitez pas à la relire ou à reprendre certains exercices pour les approfondir car la troisième partie se révèlera plus compliquée. Nous aborderons des types de données plus complexes, dits composés, comme les tableaux, les chaînes de caractères, les listes ou les pointeurs, mais aussi la programmation modulaire. Ces outils vous seront nécessaires pour aborder la programmation orientée objet dans la partie IV. Ce sera l'occasion de revenir sur certaines notions, comme les packages, que nous n'avons pas pu approfondir.

Partie 3 : Ada, les types composites

Au programme de ce chapitre : les tableaux, les chaînes de caractères, les pointeurs, les types structurés et énumérés, la lecture et écriture de fichiers, les types abstraits de données (flèches et piles), les packages mais aussi un retour sur les fonctions pour aborder la récursivité ! Bref, on commence à aborder des notions et des types de données plus complexes, qui vous demanderont plus de temps et de concentration.

Tableaux, types structurés et pointeurs constituent le cœur de cette troisième partie et ferment un fil rouge. Ces nouveaux types de données seront regroupés sous le vocable de **types composites**. Ce sera également l'occasion de nous préparer à la notion plus complexe encore de **programmation orientée objet** (ou POO) qui sera abordée lors de la partie IV. Nous effectuerons également deux TP, histoire de ne pas perdre la main : le premier consistera à créer un logiciel de gestion de bibliothèque, le second sera l'occasion de réaliser un jeu du serpent.

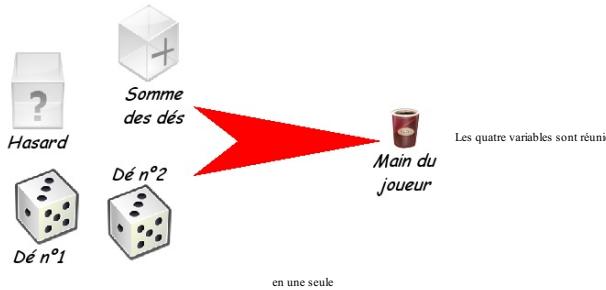
Les tableaux

Comme prévu, nous allons aborder dans ce chapitre un nouveau type de données, les tableaux ou **ARRAY** en Ada. Ce sera pour nous l'occasion d'aborder et d'expliquer ce que sont les types composites. Puis nous nous concentrerons sur notre premier type composite, les tableaux. À quoi cela ressemble-t-il en Ada ? Quel intérêt à faire des tableaux ? Comment faire des tableaux uni-, bi-, tridimensionnels ? Ce sera également l'occasion de réinvestir ce que nous avons vu dans la seconde partie, notamment sur les boucles.

Les types composites, c'est quoi ?

N'ayez crainte, vous pourrez continuer à suivre ce tutoriel, même si vous n'avez pas bien compris ce qu'est un type composite : toute la partie III y est consacrée. En fait, disons que si les types composites vont nous ouvrir de nombreuses portes qui nous étaient pour l'instant fermées, elle ne va pas pour autant révolutionner notre façon de coder ni remettre en cause ce que nous avons déjà vu.

Pour mieux comprendre, souvenons-nous de notre TP sur le craps. Nous avions beaucoup de variables à manipuler et donc à transmettre à chaque procédure ou fonction. Par exemple, il nous fallait une variable D1, une variable D2, une variable Hasard et peut-être même une variable SommeDes. Il aurait été plus pratique de ne manipuler qu'une seul variable (appelée MainJoueur ou Joueur) qui aurait contenu ces 4 variables. De même une variable composite Bourse aurait pu gérer la mise et la somme en dollars dont disposait le joueur.



Pour schématiser, les types composites ont pour but de nous faire manipuler des objets plus complexes que de simples variables. Prenons un autre exemple, nous voudrions créer un programme qui crée des fenêtres et permet également de leur donner un titre, de les agrandir ou rétrécir... dans l'optique de les réutiliser plus tard pour créer des programmes plus ambitieux. Nous devrons alors créer notamment les variables suivantes :

- X, Y : position de la fenêtre sur l'écran
- Largeur, longueur et largeur de la fenêtre
- Epais : épaisseur des bordures de la fenêtre
- Titre : le titre de la fenêtre
- Rfond, Vfond, Bfond : trois variables pour déterminer la couleur du fond de la fenêtre (Rrouge ; V: vert ; B: bleu)
- Actif : une variable pour savoir si la fenêtre est sélectionnée ou non
- Reduit : une variable pour savoir si la fenêtre est réduite dans la barre des tâches ou non
- ...

Eh oui, ça devient complexe de gérer tout ça ! Et vous avez du vous rendre compte lors du TP que cela devient de plus en plus compliqué de gérer un grand nombre de variables (surtout quand les noms sont mal trouvés comme ci-dessus). Il serait plus simple de définir un type composite **T_Fenêtre** et de ne manipuler qu'une seule variable F. Imaginons également la tête de nos fonctions :

Code : Ada

```
CreerFenetre (X=>15,
               Y=>20,
               Long=>300,
               Larg=>200,
               Epais => 2,
               Titre =>"Ma fenêtre",
               Rfond => 20,
               Vfond => 20,
               Bfond => 20,
               Actif => true,
               Reduit=> false ...)
```

Sans compter que, vous mis à part, personne ne connaît le nom et la signification des paramètres à moins de n'avoir lu tout le code (sans moi !). Imaginez également le chantier si l'on désirait créer une deuxième fenêtre (il faudra trouver de nouveaux noms de variable, Youpiii !) Il serait bien plus simple de n'avoir qu'à écrire :

Code : Ada

```
CreerFenetre (F) ;
ChangerNom (E,"Ma fenêtre") ; ...
```

Nous n'aurions pas à nous soucier de tous ces paramètres et de leur nom. C'est là tout l'intérêt des types composites : nous permettre de manipuler des objets très compliqués de la manière la plus simple possible.

Tableaux unidimensionnels

Problème

Le premier type composite que nous allons voir est le tableau, et pour commencer le tableau unidimensionnel, appelé **ARRAY** en Ada.

Quel est l'intérêt de créer des tableaux ? Pas la peine de se compliquer la vie !

Les tableaux n'ont pas été créés pour nous «compliquer la vie». Comme toute chose en informatique, ils ont été inventés pour faire face à des problèmes concrets.

Voici un exemple : en vue d'un jeu (de dé, de carte ou autre) à 6 joueurs, on a besoin d'enregistrer les scores des différents participants. D'où le code suivant :

Code : Ada

```
...
ScoreJoueur1 : natural := 0 ;
ScoreJoueur2 : natural := 0 ;
ScoreJoueur3 : natural := 0 ;
ScoreJoueur4 : natural := 0 ;
ScoreJoueur5 : natural := 0 ;
ScoreJoueur6 : natural := 0 ;
BEGIN
  ScoreJoueur1 := 15 ;
  ScoreJoueur2 := 45 ;
  ScoreJoueur3 := 27 ;
  ScoreJoueur4 := 8 ;
  ScoreJoueur5 := 19 ;
  ScoreJoueur6 := 27 ;
get(ScoreJoueur1) ; skip_line ;
get(ScoreJoueur2) ; skip_line ;
get(ScoreJoueur3) ; skip_line ;
```

```
get(ScoreJoueur4) ; skip_line ;
get(ScoreJoueur5) ; skip_line ;
get(ScoreJoueur6) ; skip_line ;
...
```

Un peu long et redondant, d'autant plus qu'il vaut mieux ne pas avoir besoin de rajouter ou d'enlever un joueur car nous serions obligés de revoir tout notre code ! Il serait judicieux, pour éviter de créer autant de variables, de les classer dans un tableau de la manière suivante :

15	45	27	8	19	27
----	----	----	---	----	----

Chaque case correspond au score d'un joueur, et nous n'avons besoin que d'une seule variable composite : Scores de type T_tableau. On dit que ce tableau est unidimensionnel car il a plusieurs colonnes mais une seule ligne !

Création d'un tableau en Ada

Déclaration

Je vous ai dit qu'un tableau se disait **ARRAY** en Ada. Mais il serait trop simple d'écrire :

Code : Ada

```
Scores : array ;
```

Car il existe plusieurs types de tableaux unidimensionnels : tableaux d'integer, tableaux de natural, tableaux de float, tableau de boolean, tableau de character... Il faut donc créer un type précis ! Et cela se fait de la manière suivante :

Code : Ada

```
type T_Tableau is array of natural ;
```

Attention ! Ce code est insuffisant !!! Il faut encore l'indexer. Pour l'heure, personne ne peut dire à partir de ce code combien de cases contient un objet de type T_Tableau, ni si la première case est bien la numéro 1 !

? HEIN ??? Qu'est-ce qu'il raconte là ?

Non je ne suis pas en plein délire, rassurez-vous. En fait, nous avons oublié lorsque nous avons écrit le tableau de numéroté les cases. Le plus évident est de le numéroté ainsi :

n°1	n°2	n°3	n°4	n°5	n°6
15	45	27	8	19	27

Il faudra donc écrire le code suivant :

Code : Ada

```
Type T_Tableau is array(1..6) of integer ;
Scores : T_Tableau ;
```

Mais le plus souvent les informaticiens auront l'habitude de numéroté ainsi :

n°0	n°1	n°2	n°3	n°4	n°5
15	45	27	8	19	27

Il faudra alors changer notre code de la façon suivante :

Code : Ada

```
Type T_Tableau is array(0..5) of integer ;
Scores : T_Tableau ;
```

La première case est donc la numéro 0 et la 6ème case est la numéro 5 ! C'est un peu compliqué, je sais, mais je tiens à l'évoquer car beaucoup de langages numérotent ainsi leurs tableaux sans vous laisser le choix (notamment le fameux langage C 😊). Bien sûr, en Ada il est possible de numéroté de 1 à 6, de 0 à 5, mais aussi de 15 à 20, de 7 à 12... il suffit pour cela de changer l'intervalle écrit dans les parenthèses.

Dernier point, il est possible en Ada d'utiliser des variables pour déclarer la taille d'un tableau, sauf que ces variables devront être... constantes !

Code : Ada

```
TailleMax : constant natural := 5 ;
Type T_Tableau is array(0..TailleMax) of integer ;
Scores : T_Tableau ;
```

Il vous sera donc impossible de créer un tableau de taille variable (pour l'instant tout du moins).

Affectation globale

Maintenant que nous avons déclaré un tableau d'entiers (Scores de type T_Tableau), encore faut-il lui affecter des valeurs. Cela peut se faire de deux façons distinctes. Première façon, la plus directe possible :

Code : Ada

```
Scores := (15,45,27,8,19,27) ;
```

Simple, efficace. Chaque case du tableau a désormais une valeur. Il est également possible d'utiliser des **agrégats**. De quoi s'agit-il ? Le plus simple est de vous en rendre compte par vous-même sur un exemple. Nous voudrions que les cases n°1 et n°3 valent 17 et que les autres valent 0. Nous pouvons écrire :

Code : Ada

```
Scores := (17,0,17,0,0,0) ;
```

Ou bien utiliser les agrégats :

Code : Ada

```
Scores := (1|3 => 17, others => 0) ;
```

Autrement dit, nous utilisons une sorte de **CASE** pour définir les valeurs de notre tableau. Autre possibilité, définir les trois premières valeurs puis utiliser un intervalle d'indices pour les dernières valeurs.

Code : Ada

```
Scores := (17,0,17,4..6 => 0) ;
```

L'ennui c'est que l'on peut être amené à manipuler des tableaux très très grands (100 cases), et que cette méthode aura vite ses limites.

Affectation valeur par valeur

C'est pourquoi on utilise souvent la seconde méthode :

Code : Ada

```
Scores(1) := 15 ;
Scores(2) := 45 ;
Scores(3) := 27 ;
Scores(4) := 8 ;
Scores(5) := 19 ;
Scores(6) := 27 ;
```

Quand nous écrivons Scores(1), Scores(2)... il ne s'agit pas du tableau mais bien des cases du tableau. Scores(1) est la case numéro 1 du tableau Scores. Cette case réagit donc comme une variable de type Integer.



Euh... c'est pas pire que la première méthode ça ?

Utilisé tel quel, si ça l'est. Mais en général, on la combine avec une boucle. Disons que nous allons initialiser notre tableau pour le remplir de 0 sans utiliser les agrégats. Voici la méthode à utiliser :

Code : Ada

```
for i in 1..6 loop
    T(i) := 0 ;
end loop ;
```



Faites attention à ce que votre boucle ne teste pas T(7) si celui-ci est indexé de 1 à 6 ! Comme T(7) n'existe pas, cela pourrait entraîner un plantage de votre programme.

L'intervalle de définition d'un type T_Tableau est très important et doit donc être respecté. Supposons que nous n'ayons pas utilisé une boucle **FOR** mais une boucle **WHILE** avec une variable i servant de compteur que nous incrémenterons nous-même de manière à tester également si la case considérée ne vaut pas déjà 0 :

Code : Ada

```
while i <= 6 and T(i) /= 0 loop
    T(i) := 0 ;
    i := i + 1 ;
end loop ;
```

La portion de code ci-dessus est censée parcourir le tableau et mettre à zéro toutes les cases non-nulles du tableau. La boucle est censée s'arrêter quand i vaut 7. Seulement que se passe-t-il lorsque i vaut 7 justement ? L'instruction **IF** teste si 7 est plus petit ou égal à 6 (Faux) et si T(7) est différent de zéro. Or T(7) n'existe pas ! Ce code va donc planter notre programme. Nous devons être plus précis et ne tester T(0) que si la première condition est remplie. Ça ne vous rappelle rien ? Alors regardez le code ci-dessous :

Code : Ada

```
while i <= 6 and then T(i) /= 0 loop
    T(i) := 0 ;
    i := i + 1 ;
end loop ;
```

Il faut donc remplacer l'instruction **AND** par **AND THEN**. Ainsi, si le premier prédictat est faux, alors le second ne sera pas testé. Cette méthode vous revient maintenant ? Nous l'avions abordée lors du chapitre sur les booléens. Peut-être sera-t-il temps d'y rejeter un œil si jamais vous avez lu les suppléments en diagonale.

Attributs pour les tableaux

Toutefois, pour parcourir un tableau, la meilleure solution est, de loin, l'usage de la boucle **FOR**. Et nous pouvons même faire encore mieux. Comme nous pourrions être amenés à modifier la taille du tableau ou sa numérotation, voici un code plus intéressant encore :

Code : Ada

```
for i in T'range loop
    T(i) := 0 ;
end loop ;
```

L'attribut **T'range** indiquera l'intervalle de numérotation des cases du tableau : 1..6 pour l'instant, ou 0..5 si l'on changeait la numérotation par exemple. Remarquez que l'attribut s'applique à l'objet T et pas à son type T_Tableau. De même, si vous souhaitez afficher les valeurs contenues dans le tableau, vous devrez écrire :

Code : Ada

```
for i in T'range loop
    put(T(i)) ;
end loop ;
```

Voici deux autres attributs liés aux tableaux et qui devraient vous servir : **T'first** et **T'last**. **T'first** vous renverra le premier indice du tableau T. **T'last** renverra le dernier indice du tableau. Si les indices de votre tableau vont de 1 à 6 alors **T'first** vaudra 1 et **T'last** vaudra 6. Si les indices de votre tableau vont de 0 à 5 alors **T'first** vaudra 0 et **T'last** vaudra 5. L'attribut **T'range** quant à lui, équivaut à écrire **T'first..T'last**.



Quel est l'intérêt de ces deux attributs puisqu'on a **T'range** ?

Il peut arriver (nous aurons le cas dans les exercices suivants), que vos boucles aient besoin de commencer à partir de la deuxième case par exemple ou de s'arrêter avant la dernière. Auquel cas, vous pourrez écrire :

Code : Ada

```
for i in T'first +1 .. T'last - k loop
    ...
```

Dernier attribut intéressant pour les tableaux : **T'length**. Cet attribut renverra la longueur du tableau, c'est à dire son nombre de cases. Cela vous évitera d'écrire **T'last - T'first +1** ! Par exemple, si nous avons le tableau suivant :

n ⁰	n ¹	n ²	n ³	n ⁴
54	98	453	45	32

Les attributs nous renverront ceci :

Attribut	Résultat
T'range	0..4
T'first	0
T(T'first)	54

T'last	4
T(T'last)	32
T'length	5

Tableaux multidimensionnels Tableaux bidimensionnels

Intérêt

Compliquons notre exercice : nous souhaitons enregistrer les scores de 4 équipes de 6 personnes. Nous pourrions faire un tableau unidimensionnel de 24 cases, mais cela risque d'être compliqué de retrouver le score du 5ème joueur de la 2ème équipe. Et c'est là qu'interviennent les tableaux à 2 dimensions ou bidimensionnels ! Nous allons ranger ces scores dans un tableau de 4 lignes et 6 colonnes.

15	45	27	8	19	27
41	5	3	11	3	54
12	13	14	19	0	20
33	56	33	33	42	65

Le 5ème joueur de la 2ème équipe a obtenu le score 3 (facile à retrouver).

Déclaration

La déclaration se fait très facilement de la manière suivante :

Code : Ada

```
type T_Tableau is array(1..4,1..6) of natural;
```

Le premier intervalle correspond au nombre de lignes, le second au nombre de colonnes.

Affectation

L'affectation directe donnerait :

Code : Ada

```
Scores := ((15,45,27,8,19,27),
           (41,5,3,11,3,54),
           (12,13,14,19,0,20),
           (33,56,33,33,42,65));
```

On voit alors que cette méthode, si pratique avec des tableaux unidimensionnels devient très lourde avec une seconde dimension. Quant à l'affectation valeur par valeur, elle exigea d'utiliser deux boucles imbriquées :

Code : Ada

```
for i in 1..4 loop
    for j in 1..6 loop
        T(i,j) := 0;
    end loop;
end loop;
```

Tableaux tridimensionnels et plus

Vous serez parfois amenés à créer des tableaux à 3 dimensions ou plus. La déclaration se fera tout aussi simplement :

Code : Ada

```
type T_Tableau3D is array(1..2,1..4,1..3) of natural;
type T_Tableau4D is array(1..2,1..4,1..3,1..3) of natural;
```

Je ne reviens pas sur l'affectation qui se fera avec trois boucles **FOR** imbriquées, l'affectation directe devenant compliquée et illisible. Ce qui est plus compliqué, c'est de concevoir ces tableaux. Voici une illustration d'un tableau tridimensionnel :

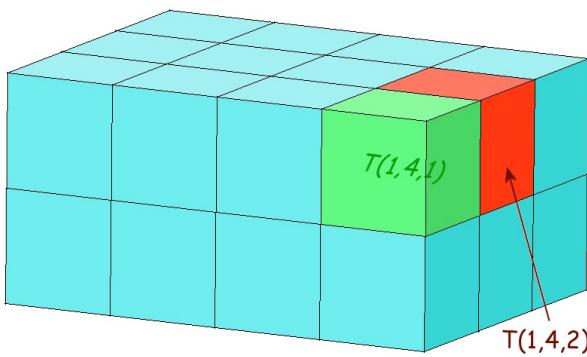
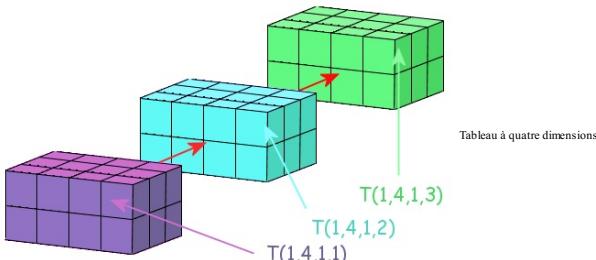


Tableau tridimensionnel

Notre tableau est en 3D, c'est un pavé droit (parallélépipède rectangle) découpé en cases cubiques. T(1,4,2) est donc le cube qui se trouve dans la première ligne, dans la 4ème colonne et dans la 2ème «rangée». Les choses se corsetent encore pour un tableau en 4 dimensions, il faut s'imaginer avoir une série de tableaux en 3D (la quatrième dimension jouant le rôle du temps) :



T(1,4,1,2) est donc le cube situé à la 1ère ligne, 4ème colonne, 1ère rangée du 2ème pavé droit.

Et mes attributs ?

C'est bien gentil de ne pas revenir sur l'affectation avec les boucles imbriquées, mais moi j'ai voulu remplir un tableau de type T_Tableau3D avec des 0, en utilisant des attributs, mais je cherche encore !

Nous avons effectivement un souci. T_Tableau3D est un type de tableau avec 2 lignes, 4 colonnes et 3 rangées, donc quand nous écrivons T'range cela correspond à l'intervalle 1 .. 2 ! Pas moyen d'avoir les intervalles 1 .. 4 ou 1 .. 3 ? Eh bien si ! Il suffit d'indiquer à notre attribut de quelle dimension nous parlons. Pour obtenir l'intervalle 1 .. 4 il faut écrire T'range(2) (l'intervalle de la seconde dimension ou le second intervalle), pour obtenir l'intervalle 1 .. 3 il faut écrire T'range(3) (l'intervalle de la troisième dimension ou le troisième intervalle). Donc T'range signifie en fait T'range(1) !

Attribut	Résultat
T'range(1)	1 .. 2
T'range(2)	1 .. 4
T'range(3)	1 .. 3
T'first(1)	1
T'first(2)	1
T'first(3)	1
T'last(1)	2
T'last(2)	4
T'last(3)	3
T'length(1)	2
T'length(2)	4
T'length(3)	3

Et mes agrégats ?

Voici une manière d'initialiser un tableau bidimensionnel de 2 lignes et 4 colonnes :

Code : Ada

```
T := (1 => (0,0,0,0), 2 => (0,0,0,0)) ;
```

On indique que la ligne 1 est (0,0,0,0) puis même chose pour la ligne 2. On peut aussi condenser tout cela :

Code : Ada

```
T := (1..2 => (0,0,0,0)) ;
```

Ou encore :

Code : Ada

```
T := (1..2 => (1..4 => 0)) ;
```

Il devient toutefois clair que les agrégats deviendront vite illisibles dès que nous souhaiterons manipuler des tableaux avec de nombreuses dimensions.

Je n'ai pas décrit ici toutes les façons d'effectuer des affectations à l'aide d'agrégats. Il est également possible de n'affecter une valeur qu'à certaines cases et pas à d'autres. Vous serez peut-être amené à utiliser ces autres agrégats. Je vous conseille alors de jeter un œil au manuel : cliquer sur *Help > Language RM*, la section *4.3.3 : Array Aggregates* devrait vous donner des indications. Et comme on dit généralement sur les forums : RIFM ! (Read This F***ing Manual = Lisez ce F***ing de Manuel !)

Des tableaux un peu moins contraints**Un type non-contraint ou presque**

Vous devriez commencer à vous rendre compte que les tableaux apportent de nombreux avantages par rapport aux simples variables. Toutefois, un problème se pose : ils sont contraints. Leur taille ne varie jamais, ce qui peut être embêtant. Reprenons l'exemple initial : nous voudrions utiliser un tableau pour enregistrer les scores des équipes. Seulement nous voudrions également que notre programme nous laisse libre de choisir le nombre de joueurs et le nombre d'équipes. Comment faire ? Pour l'heure, la seule solution est de construire un type T_Tableau suffisamment grand (500 par 800 par exemple) pour détourner quiconque de vouloir dépasser ses capacités (il faut avoir du temps à perdre pour se lancer dans une partie avec plus de 500 équipes de plus de 800 joueurs). Mais la plupart du temps, 80% des capacités seront gâchées, entraînant une perte de mémoire considérable ainsi qu'un gaspillage de temps processeur. La solution est de créer un type T_Tableau qui ne soit pas précontraint. Voici une façon de procéder avec un tableau unidimensionnel :

Code : Ada

```
type T_Tableau is array (integer range <>) of natural ;
```

Notre type T_Tableau est ainsi «un peu» plus générique.

Taurais du commencer par là ! Enfin des tableaux quasi infinis !

Attention ! Ici, c'est le type T_Tableau qui n'est pas contraint. Mais les objets de type T_Tableau devront toujours être contraints et ce, dès la déclaration. Celle-ci s'effectuera ainsi :

Code : Ada

```
T : T_Tableau(1..8) ;
--_OU
N : natural := 1065 ;
Tab : T_Tableau(5..N) ;
```

Affectation par tranche

D'où l'intérêt d'employer les attributs dans vos boucles afin de disposer de procédures et de fonctions génériques (non soumises à une taille prédefinie de tableau). Faites toutefois attention à la taille de vos tableaux, car il vous sera impossible d'écrire T := Tab ; du fait de la différence de taille entre vos deux tableaux Tab et T.

Heureusement, il est possible d'effectuer des affectations par **tranches** (**slices** en Anglais). Supposons que nous ayons trois tableaux définis ainsi :

Code : Ada

```
T : T_Tableau(1..10) ;
U : T_Tableau(1..6) ;
V : T_Tableau(1..4) ;
```

Il sera possible d'effectuer nos affectations de la façon suivante :

Code : Ada

```
T(1..6) := U ;
T(7..10) := V ;
```

Où même, avec les attributs :

Code : Ada

```
T('first..T'first+U'length-1) := U ;
T(T'first+U'length..T'first+U'length+V'length-1) := V ;
```



L'affectation par tranche n'est pas possible pour des tableaux multidimensionnels !

Déclarer un tableau en cours de programme

Mais cela ne répond toujours pas à la question : «Comment créer un tableau dont la taille est définie par l'utilisateur ?». Pour cela, vous allez devoir sortir temporairement du cadre des tableaux et vous souvenir de ce dont nous avions parler dans le chapitre sur les variables et leur déclaration. Lors des compléments, nous avions vu le bloc de déclaration. L'instruction **DECLARE** va nous permettre de déclarer des variables en cours de programme, et notamment des tableaux. Voici un exemple :

Code : Ada

```
procedure main is
    n : Integer;
    type T_Tableau is array (integer range <>) of integer;
begin
    Put("Quelle est la longueur du tableau ? ");
    get(n); skip_line;
    declare
        tab : T_Tableau(1..n);
    begin
        --instructions ne nous intéressent pas
    end;
end main;
```

Ainsi nous saisissons une variable **n**, puis avec **DECLARE**, nous ouvrons un bloc de déclaration nous permettant de créer notre tableau **tab** à la longueur souhaitée. Puis, il faut terminer les déclarations et ouvrir un bloc d'instruction avec **BEGIN**. Ce nouveau bloc d'instructions se terminera avec le **END** .



Mon objet **tab** disparaîtra avec l'instruction **END** ; à la fin du bloc **DECLARE** ! Inutile de utiliser entre **END** ; et **END main** ;

Quelques exercices

Voici quelques exercices pour mettre en application ce que nous venons de voir.

Exercice 1

Énoncé

Créez un programme Moyenne qui saisit une série de 8 notes sur 20 et calcule la moyenne de ces notes. Il est bien sûr hors de question de créer 8 variables ni d'écrire une série de 8 additions car le programme devra pouvoir être modifié très simplement pour pouvoir calculer la moyenne de 7, 9, 15... notes, simplement en modifiant une variable !

Solution

Secret (cliquez pour afficher)

Code : Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;

procedure Moyenne is
    Nb_Notes : constant integer := 8;          --
    Constante donnant la taille des tableaux
    type T_Tableau is array(1..Nb_Notes) of float;  --
    type tableau pour enregistrer les notes
    Notes : T_Tableau;                          --
    tableau de notes
    Moy : float;                                --
    variable servant à enregistrer la moyenne

    function saisie return T_Tableau is           --
    fonction effectuant la saisie des valeurs d'un tableau
        T : T_Tableau;
    begin
        for i in T'range loop
            Put("Entrez la "); Put(i,1);
            if i=1
                then put("ère note : ");
                else put("ème note : ");
            end if;
            get(T(i)); skip_line;
        end loop;
        return T;
    end saisie;

    function somme(T:T_Tableau) return float is      --
    fonction effectuant la somme des valeurs contenues dans un tableau
        S : float := 0.0;
    begin
        for i in T'range loop
            S:= S + T(i);
        end loop;
        return S;
    end somme;

    begin
        Procedure principale
        Notes := saisie;
        Moy := Somme(Notes)/float(Nb_Notes);
        Put("La moyenne est de "); Put(Moy);
    end Moyenne;
```



J'ai un problème dans l'affichage de mon résultat ! Le programme affiche 1.24000E+01 au lieu de 12.4 !

Par défaut, l'instruction **Put()** affiche les nombres flottants sous forme d'une écriture scientifique (le E+01 remplaçant une multiplication par dix puissance 1). Pour régler ce problème, vous n'aurez qu'à écrire :

Code : Ada

```
Put(Moy,2,2,0);
```

Que signifient tous ces paramètres ? C'est simple, l'instruction **put()** s'écrit en réalité ainsi :

```
Ada.Float_Text_IO.Put(Item=>Moy, Fore => 2, Aft => 2, Exp => 0);
```

- L'instruction **Put** utilisée ici est celle provenant du package **Ada.Float_Text_IO**.
- Le paramètre **Item** est le nombre de type **float** qu'il faut afficher.
- Le paramètre **Fore** est le nombre de chiffres placés avant la virgule qu'il faut afficher (**Fore** pour **Before** = Avant). Cela évite d'afficher d'immenses blancs devant vos nombres.
- Le paramètre **Aft** est le nombre de chiffres placés après la virgule qu'il faut afficher (**Aft** pour **After** = Après).
- Le paramètre **Exp** correspond à l'exposant dans la puissance de 10. Pour tous ceux qui ne savent pas (ou ne se souviennent pas) ce qu'est une puissance de 10 ou une écriture scientifique, retenez qu'en mettant 0, vous n'aurez plus de E à la fin de votre nombre.

En écrivant **Put(Moy,2,2,0)** nous obtiendrons donc un nombre avec 2 chiffres avant la virgule, 2 après et pas de E+01 à la fin.

Exercice 2

Énoncé

Rédigez un programme Pascal qui affichera le triangle de Pascal jusqu'au rang 10. Voici les premiers rangs :

Code : Console

```
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Ce tableau se construit de la manière suivante : pour calculer une valeur du tableau, il suffit d'additionner celle qui se trouve au dessus et celle qui se trouve au-dessus à gauche. Le triangle de Pascal est très utile en [Mathématiques](#).

Comme on ne peut créer de tableaux «triangulaires», on créera un tableau «carré» de taille 10 par 10 rempli de zéros que l'on n'affichera pas. On prendra soin d'afficher les chiffres des unités les uns en dessous des autres.

Solution

Secret ([cliquez pour afficher](#))

Code : Ada

```
with Ada.Text_IO,Ada.Integer_Text_IO ;
use Ada.Text_IO,Ada.Integer_Text_IO ;

procedure Pascal is
    Taille : constant integer := 10 ;
    type T_Tableau is array(1..Taille, 1..Taille) of integer ;
    Pascal : T_Tableau ;
begin
    function init return T_Tableau is
        T : T_Tableau ;
    begin
        T := (others => 0) ;
        return T ;
    end init ;
    function CreerTriangle (Tab : T_Tableau) return T_Tableau is
        T : T_Tableau := Tab ;
    -- comme on ne peut pas modifier le paramètre, on en crée une copie
    begin
        for i in T'range loop
            T(i,1) := 1 ;
        -- le premier nombre vaut toujours 1
        for j in 2..i loop
            -- on remplit ensuite la ligne à partir du deuxième nombre
            T(i,j) := T(i-1,j) + T(i-1,j-1) ;
        end loop ;
    end;
    return T ;
    end CreerTriangle ;
    Procedure Afficher(T : T_Tableau) is
    begin
        for i in T'range loop
            for j in 1..i loop
                Put(T(i,j),4) ;
            end loop ;
            new_line ;
        end loop ;
    end Afficher ;
begin
    Pascal := init ;
    Pascal := CreerTriangle(Pascal) ;
    Afficher(Pascal) ;
end Pascal ;
```

Pourquoi dans ta fonction CreerTriangle tu n'as pas écrit ça ?

Code : Ada

```
for i in T'range loop
    for j in 1..i loop
        -- on remplit ensuite la ligne à partir du deuxième nombre
        if j = 1 then
            T(i,j) := 1 ;
        else
            T(i,j) := T(i-1,j) + T(i-1,j-1) ;
        end if ;
    end loop ;
end loop ;
```

Surtout pas ! Car à chaque itération (chaque tour de boucle) il aurait fallu faire un test, ou un test coûte de la place en mémoire et surtout du temps processeur. Cela aurait fait perdre beaucoup d'efficacité à notre programme alors que le cas de figure testé (savoir si l'on est dans la première colonne) est relativement simple à gérer et ne nécessite pas de condition. De même pour l'affichage, je n'ai pas testé si les nombres valaient 0, il suffit d'amener la boucle avant de les atteindre, et ça, ça se calcule facilement.

Il est également possible pour améliorer ce code, de fusionner les fonctions init et CreerTriangle en une seule, de la manière suivante :

Code : Ada

```
function init return T_Tableau is
begin
    for i in T'range loop
        T(i,1) := 1 ; -- le premier nombre vaut toujours 1
        for j in 2..i loop
            -- on remplit ensuite la ligne à partir du deuxième nombre
            T(i,j) := T(i-1,j) + T(i-1,j-1) ;
        end loop ;
        for j in i+1 .. T'last loop -- on complète avec des 0
            T(i,j) := 0 ;
        end loop ;
    end loop ;
    Return T ;
end init ;
```

Exercice 3

Énoncé

Réalisez un programme TriTableau qui crée un tableau unidimensionnel avec des valeurs entières aléatoires, l'affiche, trié ses valeurs par ordre croissant et enfin affiche le tableau une fois trié.

Solution

Secret ([cliquez pour afficher](#))

Code : Ada

```
With ada.Text_IO, Ada.Integer_Text_IO,Ada.Numerics.Discrete_Random ;
Use ada.Text_IO, Ada.Integer_Text_IO ;
```

```

Procedure TriTableau is
    Taille : constant natural := 12 ;
    Type T_Tableau is array(1..Taille) of integer ;
    --init renvoie un tableau à valeurs aléatoires
    function init return T_Tableau is
        T:T_Tableau ;
        subtype Intervalle is integer range 1..100 ;
        package Aleatoire is new Ada.numerics.discrete_random(Intervalle) ;
        use Aleatoire ;
        Hasard : generator ;
    begin
        Reset(Hasard) ;
        for i in T'range loop
            T(i) := random(Hasard) ;
        end loop ;
        return T ;
    end init ;
    --Afficher affiche les valeurs d'un tableau sur une même ligne
    procedure Afficher(T : T_Tableau) is
    begin
        for i in T'range loop
            Put(T(i),4) ;
        end loop ;
    end Afficher ;
    --Echanger est une procédure qui échange deux valeurs : a vaudra b et b va
    procedure Echanger(a : in out integer ; b : in out integer) is
        c : integer ;
    begin
        c := a ;
        a := b ;
        b := c ;
    end Echanger ;
    --
    RangMin cherche la valeur minimale dans une partie d'un tableau ; du rang deb
    --Elle ne renvoie pas le minimum mais son rang, son indice dans le tableau
    function RangMin(T : T_Tableau ; debut : integer ; fin : integer) return integer
    Rang : integer := debut ;
    Min : integer := T(debut) ;
    begin
        for i in debut..fin loop
            if T(i)<Min then Min := T(i) ;
        end if ;
        end loop ;
        return Rang ;
    end RangMin ;
    --Trier est une fonction qui renvoie un tableau trié du plus petit au plus
    --Principe, elle cherche la valeur minimale du tableau et l'échange avec la pre
    --puis elle cherche le minimum dans le tableau à partir de la deuxième valeur e
    --avec la première valeur, puis elle recommence avec la troisième valeur..
    function Trier(Tab : T_Tableau) return T_Tableau is
        T : T_Tableau := Tab ;
    begin
        for i in T'range loop
            Echanger(T(i),T(RangMin(T,i,T'last))) ;
        end loop ;
        return T ;
    end Trier ;
    T : T_Tableau ;
begin
    T := init ;
    Put_line("Le tableau généré par l'ordinateur est le suivant :") ;
    Afficher(T) ; New_line ;
    Put_line("Voici le tableau, une fois trié : ") ;
    Afficher(Trier(T)) ;
end TriTableau ;

```

L'exemple que je vous donne est un algorithme de tri par sélection. C'est disons l'un des algorithmes de tri les plus barbares. C'est la méthode la plus logique à première vue, mais sa complexité n'est pas vraiment intéressante (par complexité, on entend le nombre d'opérations et d'itérations nécessaires et pas le fait que vous ayez compris ou pas). Nous verrons plus tard d'autres algorithmes de tri plus efficaces. En attendant, si ce code vous semble compliqué, je vous invite à lire le tutoriel de K-Phone sur le tri par sélection : lui, il a cherché le maximum et pas le minimum, mais le principe est le même ; en revanche, son code est en C, mais cela peut être l'occasion de voir par vous-même les différences et ressemblances entre l'Ada et le C.

Nous reviendrons durant la quatrième partie sur la complexité des algorithmes et nous en profiterons pour aborder des algorithmes de tri plus puissants. Mais avant cela, vous avez beaucoup de pain sur la planche ! 😊

Pour les utilisateurs de la norme Ada2012

Ce que nous venons de voir est valable pour tous les compilateurs, quelle que soit la norme Ada utilisée. En revanche, ce qui va suivre n'est valable que pour les compilateurs intégrant la norme Ada2012. Comme nous l'avions déjà vu lors du chapitre sur les conditions, la nouvelle norme internationale du langage Ada, appelée Ada2012, s'est inspirée des langages fonctionnels. C'est vrai pour les expressions conditionnelles, mais c'est également vrai pour les boucles. Ainsi, la boucle FOR a été enrichie de nouvelles fonctionnalités. La boucle FOR que vous connaissez devrait désormais être nommée FOR IN puisque s'ajoutent désormais trois nouvelles boucles : FOR OF, FOR ALL et FOR SOME.

Boucle FOR OF

La boucle FOR OF est un vrai coup de pouce pour les programmeurs. Aussi appelée **foreach** dans bon nombre de langages, va appliquer vos instructions à tous les éléments de votre tableau (of = de) sans que vous n'ayez à vous tracasser de leurs indices. Prenons un exemple :

Code : Ada

```

Procedure Augmente is
    Type T_Tableau is array(1..8) of integer ;
    T : T_Tableau := (7,8,5,4,3,6,9,2) ;
begin
    for E of T loop
        E := E + 1 ;
    end loop ;
end Augmente ;

```

Le programme ci-dessus se contente d'incrémenter chacune des valeurs d'un tableau. Mais au lieu d'utiliser une boucle FOR IN comme d'habitude nous utilisons une boucle FOR OF. La variable E n'a pas besoin d'être déclarée, elle est automatiquement créée par la boucle fonction du type d'éléments contenus dans le tableau T. Vous devez comprendre la phrase « FOR E OF T » ainsi : « Pour chaque élément E de T, faire ceci ou cela ». L'avantage, c'est que nous ne risquons plus de nous tromper dans les indices, Ada prend automatiquement chacun des éléments du tableau. Et si jamais vous aviez un doute sur le typeage de E, il est possible de réécrire la boucle ainsi :

Code : Ada

```

for E : Integer of T loop
    E := E + 1 ;
end loop ;

```

Encore mieux : la même boucle peut s'appliquer à des tableaux multidimensionnels.

Code : Ada

```

Procedure Augmente is
    Type T_Tableau is array(1..13,1..15,1..32,1..9754) of integer ;
    T : T_Tableau ;
begin
    --Initialisez cet énorme tableau vous-même ;-
    for E of T loop

```

```
E := E + 1 ;
end loop ;
end Augmente ;
```

Comme vous pouvez le constater, le type `T_Tableau` ci-dessus est vraiment abominable : très grand et de dimension 4. Normalement, cela nous obligerait à imbriquer quatre boucles `FOR IN` avec le risque de mélanger les intervalles. Avec la boucle `FOR OF`, nous n'avons aucun changement à apporter et aucun risque à prendre : Ada2012 gère tout seul.

Expressions quantifiées

Les deux autres boucles (`FOR ALL` et `FOR SOME`) ne sont utilisables que pour des expressions, appelées **expressions quantifiées**. Autrement dit, elles permettent de parcourir notre tableau afin d'y effectuer des tests.

Quantificateur universel

Première expression quantifiée : la boucle `FOR ALL`, aussi appelée **quantificateur universel**. Celle-ci va nous permettre de vérifier que **toutes** les valeurs d'un tableau répondent à un critère précis. Par exemple, nous souhaiterions savoir si tous les éléments du tableau `T` sont positifs :

Code : Ada

```
...
Tous_Positifs : Boolean ;
BEGIN
  ...
  Tous_Positifs := (FOR ALL i IN T'range => T(i) > 0) ;
  ...

```

Pour que la variable `Tous_Positifs` vaille `TRUE`, tous les éléments de `T` doivent être positifs. Si un seul est négatif ou nul, alors la variable `Tous_Positifs` vaudra `FALSE`. Mais ce code peut encore être amélioré en le combinant avec la boucle `FOR OF`:

Code : Ada

```
...
Tous_Positifs : Boolean ;
BEGIN
  ...
  Tous_Positifs := (FOR ALL e OF T => e > 0) ;
  ...

```

L'expression anglaise « *for all* » peut se traduire en « *pour tout* » ou « *Quel que soit* », ce qui est un quantificateur en Mathématiques, noté \forall . Exprimé à l'aide d'opérateurs booléens, cela équivaudrait à : « $(T(1)>0) \text{ AND } (T(2)>0) \text{ AND } (T(3)>0) \text{ AND } (T(4)>0) \text{ AND } \dots$ ».

Quantificateur existentiel

Poussons le vice encore plus loin. Tant que tous les éléments ne seront pas positifs, nous incrémenterons tous les éléments du tableau. Autrement dit, tant qu'il existe un élément négatif ou nul, on incrémentera tous les éléments. Cette dernière phrase, et notamment l'expression « *il existe* », se note de la façon suivante :

Code : Ada

```
(FOR SOME i IN T'range => T(i) <= 0)
-- OU
(FOR SOME e OF T => e <= 0)
```

Il s'agit là du **quantificateur existentiel**. Il suffit qu'un seul élément réponde au critère énoncé pour que l'expression soit vraie. Nous pouvons désormais compléter notre code :

Code : Ada

```
WHILE (FOR SOME e OF T => e <= 0) LOOP
  FOR e OF T LOOP
    e := e + 1 ;
  END LOOP ;
END LOOP ;
```

L'expression anglaise « *for some* » peut se traduire en « *pour quelques* ». Le quantificateur mathématique correspondant est « *il existe* » et se note \exists . Exprimé à l'aide d'opérateurs booléens, cela équivaudrait à : « $(T(1)\leq 0) \text{ OR } (T(2)\leq 0) \text{ OR } (T(3)\leq 0) \text{ OR } (T(4)\leq 0) \text{ OR } \dots$ ».

Maintenant que vous connaissez les tableaux, notre prochain chapitre portera sur un type similaire : les chaînes de caractères ou string (en Ada). Je ne saurais trop vous conseiller de vous exercer à l'utilisation des tableaux car nous serons amenés à les utiliser très souvent par la suite, et le prochain chapitre ne s'en éloignera guère. Donc s'il vous reste des questions, des zones d'ombre, relisez ce cours et effectuez les exercices. N'hésitez pas non plus à vous fixer des projets ou des TP personnels.

En résumé :

- Pour utiliser des tableaux, vous devez créer un type spécifique en indiquant la taille, le nombre de dimensions et le type des éléments du tableau.
- La manipulation des tableaux se fait généralement à l'aide de boucles afin de traiter les divers éléments. Vous utiliserez le plus souvent la boucle `FOR` et, si votre compilateur est compatible avec la norme Ada2012, la boucle `FOR OF`.
- Un tableau est un type composite contraint. Il peut contenir autant d'informations que vous le souhaitez mais vous devez définir sa taille au préalable. Une fois le tableau déclaré, sa taille ne peut plus varier.

Les chaînes de caractères

Après les tableaux, voici un nouveau type composé Ada : les chaînes de caractères ou strings. Rentrer les langues les garçons, il n'y aura rien de sexuel dans ce chapitre. À dire vrai, vous avez déjà manipulé des strings par le passé (je veux dire, en Ada) et ce chapitre ne fera que mettre les choses au clair car vous disposez dès maintenant de toutes les connaissances nécessaires.

Présentation des Chaînes de Caractères

Souvenez-vous, vous avez déjà manipulé des `string` ou chaînes de caractères par le passé. Notamment lorsque vous écriviez :

Code : Ada

```
Put("Bonjour !");
```

Le texte entre guillemets "Bonjour !" est un `string`.



Ah... c'est pas ce que j'avais pensé... ☺ On va faire tout un chapitre dessus ?

Ce chapitre ne sera pas long. Nous avons toutes les connaissances utiles, il s'agit juste de découvrir un type composite que nous utilisons depuis longtemps sans nous en rendre compte. Qu'est-ce que finalement qu'une chaîne de caractères ? Eh bien le string ci-dessus se résume en fait à ceci :

n°1	n°2	n°3	n°4	n°5	n°6	n°7	n°8	n°9
'B'	'o'	'n'	'j'	'u'	't'	' '	'!	

Eh oui ! Ce n'est rien d'autre qu'un tableau de caractères indexé à partir de 1. Notez que certains langages comme le célèbre C, commencent leurs indices à 0 et ajoutent un dernier caractère à la fin : le caractère de fin de texte. Ce n'est pas le cas en Ada où le type `String` est très aisément manipulable.

Déclaration et affectation d'un string

Nous allons créer un programme `ManipString`. Nous aurons besoin du package `ada.text_io`. De plus, avant de créer des instructions, nous allons devoir déclarer notre `string` de la manière suivante :

Code : Ada

```
with ada.text_io;
use ada.text_io;

procedure ManipString is
  txt : string(1..6);
begin
  txt := "Salut!";
  put(txt);
end ManipString;
```

Eh oui ! Un `String` n'est rien d'autre qu'un tableau de caractères, il est donc contraint : son nombre de cases est limité et doit être indiqué à l'avance. Par conséquent, il est possible de lui affecter la chaîne de caractères "Salut!" mais pas la chaîne "Salut !" qui comporte 7 caractères (n'oubliez pas qu'un espace est le caractère ' '). Il est aussi possible de déclarer notre objet `txt` tout en lui affectant une valeur initiale :

Code : Ada

```
with ada.text_io;
use ada.text_io;

procedure ManipString is
  txt : string := "Salut!";
begin
  put(txt);
end ManipString;
```

L'objet `txt` est automatiquement indexé de 1 à 6.

Quelques opérations sur les strings

Accès à une valeur

Comme pour un tableau, il est possible d'accéder à l'une des valeurs contenues dans le `String`. Par exemple :

Code : Ada

```
with ada.text_io;
use ada.text_io;

procedure ManipString is
  txt : string := "Bonjour jules !";
  indice : integer;
begin
  put_line(txt);
  for i in txt'range loop
    if txt(i) = 'e' then
      txt(indice) := i;
    end if;
  end loop;
  txt(indice) := 'i';
  txt(indice+1) := 'e';
  put("oops. ");
  put(txt);
end ManipString;
```

Ce programme cherche la dernière occurrence de la lettre 'e' et la remplace par un 'i'. Il affiche ainsi :

Code : Console

```
Bonjour Jules !
Oops. Bonjour Julie !
```

Remarquez au passage qu'il est toujours possible d'utiliser les attributs.

Accès à plusieurs valeurs

Nous voudrions maintenant modifier "Bonjour Jules !" en "Bonsoir Julie !". Cela risque d'être un peu long de tout modifier lettre par lettre. Voici donc une façon plus rapide :

Code : Ada

```
with ada.text_io;
use ada.text_io;

procedure ManipString is
  txt : string := "Bonjour Jules !";
begin
  put_line(txt);
  txt(4..7) := "soir";
  for i in txt'range loop
    if txt(i) = 'e' then
      txt(i) := '!';
      txt(i+1) := 'e';
      exit;
    end if;
  end loop;
  put("oops. ");
  put(txt);
end ManipString;
```

Il est ainsi possible d'utiliser les slices (tranches) pour remplacer tout un paquet de caractères d'un coup. Vous remarquerez que j'en ai également profité pour condenser mon code et supprimer la variable `indice`. Personne n'est parfait : il est rare d'écrire un code parfait dès la première écriture. La plupart du temps, vous tâtonnerez et peu à peu, vous parviendrez à supprimer le superflu.

de vos codes.

Modifier la casse

Autre modification possible : modifier la casse (majuscule/minuscule). Nous aurons besoin pour cela du package `Ada.characters.handling`. Le code suivant va ainsi écrire "Bonjour" en majuscule et transformer le 'j' de Jules en un 'J' majuscule.

Code : Ada

```
with ada.text_io, Ada.characters.handling ;
use ada.text_io, Ada.characters.handling ;

procedure ManipString is
    txt : string := "Bonjour jules !";
begin
    put_line(txt);
    txt(1..7) := to_upper(txt(1..7));
    for i in txt'range loop
        if txt(i) = 'j' then
            txt(i) := 'J';
        elsif txt(i) = 'e' then
            txt(i) := to_upper(txt(i));
        end if;
    end loop;
    put("oops.");
    put(txt);
end ManipString;
```

La fonction `To_Upper()` transforme un caractère ou un string pour le mettre en majuscule. Il existe également une fonction `To_Lower()` qui met le texte en minuscule.

Concaténation

Nous allons maintenant écrire un programme qui vous donne le choix entre plusieurs prénom (entre plusieurs strings) et affiche ensuite une phrase de bienvenue. Les différents prénoms possibles seront : "Jules", "Paul" et "Frédéric". Contrainte supplémentaire : il faut utiliser le moins d'instructions Put pour saluer notre utilisateur.

Code : Ada

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure ManipString is
    txt : string := "Bienvenue, ";
    Nom_1 : string := "Jules";
    Nom_2 : string := "Paul";
    Nom_3 : string := "Frédéric";
    choix : integer;
begin
    Put_line("Comment vous appelez-vous ?");
    Put_line("1 - " & Nom_1);
    Put_line("2 - " & Nom_2);
    Put_line("3 - " & Nom_3);
    Get(choix); skip_line;
    case choix is
        when 1 => Put(txt & Nom_1);
        when 2 => Put(txt & Nom_2);
        when others => Put(txt & Nom_3);
    end case;
end ManipString;
```

Nous avons utilisé le symbole & (appelé esperluette) pour effectuer une opération de **concaténation**. Cela signifie que l'on met bout à bout deux strings (ou plus) pour en obtenir un plus long.

Attention ! Les strings obtenus après ces concaténations ont des longueurs variables. Vous ne pourrez donc pas écrire le code suivant :

Code : Ada

```
case choix is
    when 1 => txt := txt & Nom_1;
    when 2 => txt := txt & Nom_2;
    when others => txt := txt & Nom_3;
end case;
```

Eh oui ! Le string `txt` a été initialisé avec une longueur de 11 caractères. Et comme tout tableau, sa longueur ne peut être modifiée ou variable. Or « `txt & Nom_1` » a une longueur de 17 caractères (11+6) et « `txt & Nom_2` » a une longueur de 16 (11+5).

Transformer une variable en string et inversement

Transformer une variable en string



Et si au lieu de "Bienvenue #####", je veux afficher une ligne disant "Vous avez fait le choix n°#, vous vousappelez donc #####", je peux concaténer des strings et des variables de types integer ?

Euh... non ! Rappelez-vous, je vous ai dit que le langage Ada avait un fort typage, autrement dit on ne mélange pas les choux et les carottes, les strings et les integer ! Par contre, il y a un moyen de parvenir à votre but en respectant ces contraintes de fort typage, grâce aux... attributs (eh oui encore) !

Code : Ada

```
Put("Vous avez fait le choix n"
" & integer'image(choix) & ", vous vousappelez donc " & Nom_1);
```

L'attribut `integer'image()` renvoie un string, une «`image`» de l'integer placé entre parenthèses. Bien entendu, cet attribut n'est pas réservé au type Integer et s'applique à tous types de variables.

Transformer un string en variable



Et l'inverse ? Si je veux transformer un string en variable ?

Eh bien tout dépend du type de variable souhaité. Si vous voulez une variable du type integer, alors vous pourrez utiliser l'attribut `Integer'value()`. Entre les parenthèses, vous n'aurez qu'à écrire votre string et l'attribut `'value` le transformera en un integer (ou un natural, ou un float...).

Comparaison

Oui, il est possible de comparer des strings ! ☺ Il est possible de tester leur égalité avec l'opérateur = ; il est aussi possible d'utiliser les opérateurs d'ordre <, >, <=, >= !



Et il compare quoi le programme ? Ça veut dire quoi si j'écris "Avion" > "Bus" ?

Les opérateurs d'ordres utilisent l'ordre alphabétique et sont insensibles à la casse (Majuscule/Minuscule). La comparaison "Avion" > "Bus" devrait donc renvoyer FALSE, à ceci près que le compilateur ne voudra pas que vous l'écriviez tel quel. Il préférera :

Code : Ada

```
...
txt1 : string := "Avion" ;
txt2 : string := "Bus" ;
BEGIN
  if txt1 > txt2
    then ...
  end if;
```

Saisie au clavier

Et si je souhaite saisir un nom directement au clavier ?

Alors il existe une fonction pour cela : `get_line`. Elle saisira tout ce que vous tapez au clavier jusqu'à ce que vous appuyiez sur Entrée (donc pas besoin d'utiliser `skip_line` par la suite) et renverra ainsi une chaîne de caractères :

Code : Ada

```
txt := get_line ;
```



La longueur du string renvoyé par `get_line` est inconnue ! Or celle du string `txt` est connue et fixe. Si le texte saisi est trop long, vous vous exposez à des erreurs, s'il est trop court et que `txt` n'a pas été initialisé (en le remplaçant de "" par exemple), vous risquez des erreurs lors de l'affichage. Contraintain n'est-ce pas ?

Chaines de caractères non contraintes**Déclarer des strings illimités !**

Il existe une façon de gérer des chaînes de caractères sans être limité par leur longueur. Pour cela, vous aurez besoin du package `Ada.Strings.Unbounded`. Et nous déclarerons notre variable ainsi :

Code : Ada

```
txt : unbounded_string ;
```

Cela signifie que `txt` sera une «chaîne de caractère illimitée» !!!



Wouhouh ! J'essaye ça tout de suite ! Taurais du commencer par là !!

Attendez ! Les `unbounded_string` ne s'utilisent pas de la même manière que les `string` normaux ! Simplement parce que leur nature est différente. Lorsque vous déclarez un tableau, par exemple :

Code : Ada

```
txt : string(1..8) ;
```

L'ordinateur va créer en mémoire l'équivalent de ceci :

1	2	3	4	5	6	7	8
?	?	?	?	?	?	?	?

Lorsque vous déclarez un `unbounded_string` comme nous l'avons fait tout à l'heure, l'ordinateur créera ceci :

```
{}
```

C'est en fait une liste, vide certes mais une liste tout de même. À cette liste, le programme ajoutera des caractères au fur et à mesure. Il n'est plus question d'indices, ce n'est plus un tableau. Nous verrons les listes après avoir vu les pointeurs, ce qui vous éclairera sur les limites du procédé.

Opérations sur les `unbounded_string`

Nous pouvons ensuite affecter une valeur à nos `unbounded_string`. Seulement il est impossible d'écrire ceci :



```
...
txt : unbounded_string ;
BEGIN
  txt := "coucou !" ;
...
```

Cela reviendrait à affecter un String à un objet de type `unbounded_string` ! Or nous savons que Ada est un langage à fort type ! Un `Unbounded_String` ne peut recevoir comme valeur un `string` !

Vous devrez donc utiliser les fonctions `To_Unbounded_String()` et `To_String()` pour convertir de la manière suivante :

Code : Ada

```
...
txt : unbounded_string ;
BEGIN
  txt := To_Unbounded_String("coucou !");
  put(To_String(txt)) ;
...
```

Notre objet `txt` aura alors l'allure suivante :

```
{'c','o','u','c','o','u',' ','!'};
```

Il est toutefois possible d'opérer des concaténations grâce à l'opérateur `&`. Mais cette fois, plusieurs opérateurs `&` ont été créés pour pouvoir concaténer :

- un `string` et un `string` pour obtenir un `string`
- un `unbounded_string` et un `string` pour obtenir un `unbounded_string`
- un `string` et un `unbounded_string` pour obtenir un `unbounded_string`
- un `unbounded_string` et un `unbounded_string` pour obtenir un `unbounded_string`
- un caractère et un `unbounded_string` pour obtenir un `unbounded_string`
- un `unbounded_string` et un caractère pour obtenir un `unbounded_string`

Nous pouvons ainsi écrire :

Code : Ada

```
txt := txt & "Comment allez-vous ?"
```

Notre objet `txt` ressemblera alors à ceci :

```
{'c','o','u','c','o','u',' ','!','&','C','o','u','s','?'}
```



Pourquoi tu ne présentes pas ça sous la forme d'un tableau, ce serait plus clair et on n'aurait pas besoin de compter pour connaître l'emplacement d'un caractère ?

 N'oubliez pas que `txt` est un `unbounded_string`, pas un tableau (ni un `string`) ! C'est une liste, et il n'est pas possible d'écrire `txt(4)` ou `txt(1..3)` ! Ces écritures sont réservées aux tableaux (les `strings` sont des tableaux ne l'oublions pas) et sont donc fausses concernant notre objet `txt` ! Pour tout vous dire, un `Unbounded_String` n'a accès qu'à la première lettre. Celle-ci donne accès à la seconde, qui elle-même donne accès à la troisième et ainsi de suite.

En effet, de nombreuses fonctionnalités liées aux tableaux ne sont plus disponibles. Voici donc quelques fonctions pour les remplacer :

Code : Ada

```
n := length(txt);           --  
la variable n (natural) prendra comme valeur la  
--longueur de l'unbounded_string txt  
  
char := Element(txt,5);     --  
la variable char (character) prendra la 5ème valeur  
--de l'unbounded_string txt  
  
Replace_Element(txt,6,char); --  
le 6ème élément de txt est remplacé par le caractère char  
  
txt := Null_Unbounded_String; --  
txt est réinitialisé en une liste vide
```

Une indication tout de même, si vous ne souhaitez utiliser qu'une certaine partie de votre `unbounded_string` et que vous voudriez écrire `txt(5..8)`, vous pouvez vous en sortir en utilisant `To_String()` et `To_Unbounded_String()` de la manière suivante :

Code : Ada

```
put(to_string(txt)(5..8));  
--OU  
txt:= to_unbounded_string(to_string(txt)(5..8));  
put(to_string(txt));
```

Comme vous pouvez vous en rendre compte, la manipulation des `unbounded_strings` est plus lourde que celle des `strings`. Voilà pourquoi nous préférerons utiliser en général les `strings`, nous réservons l'utilisation des `unbounded_strings` aux cas d'extrêmes urgences. ☺

Nous voilà à la fin de ce chapitre sur les `strings` et les `unbounded_strings`. Cela constitue également un bon exercice sur les tableaux. Mais ne vous croyez pas sortis d'affaire : nous n'en avons pas terminé avec les tableaux. Les prochains chapitres, consacrés à la **programmation modulaire** (l'utilisation et la création de packages) ou à la manipulation des fichiers, seront l'occasion de travailler encore avec les tableaux, les `strings` ou les `unbounded_strings`.

En résumé :

- Un `String` n'est rien de plus qu'un tableau unidimensionnel composé de caractères.
- Par conséquent, si vous déclarer un `String`, vous devez absolument définir sa taille : il s'agit d'un type contraint.
- Les `Strings` supportent les tests d'égalité et d'inégalité. Ces derniers comparent en fait leur classement dans l'ordre alphabétique. À cela s'ajoute l'opération de concaténation, symbolisée par l'espérance &, qui permet de mettre bout à bout deux chaînes de caractère.
- Préférez toujours les `String` pour leur simplicité. Si réellement la contrainte de taille devient un handicap, optez alors pour un `Unbounded_String`. Vous aurez alors besoin du package `Ada.Strings.Unbounded`.

La programmation modulaire I : les packages

Pour ce nouveau chapitre, nous allons revenir sur les tableaux.

Encore des tableaux ? Avec les strings, ça va faire le troisième chapitre !

Rassurez-vous, les tableaux ne constitueront qu'un prétexte et non pas le cœur de ce chapitre. Nous avons vu tout ce qu'il y avait à voir sur les tableaux. Mais cela ne doit pas nous faire oublier leur existence, bien au contraire. Nous allons très régulièrement les utiliser. Mais, si vous avez effectué les exercices sur les tableaux, vous avez du vous rendre compte qu'il fallait très régulièrement décrire les mêmes fonctions, les mêmes procédures, les mêmes déclarations de type... Se pose alors la question :

Ne serait-il pas possible de mettre tout cela dans un fichier réutilisable à l'envers ?

Et la réponse, vous en doutez, est oui ! Ces fichiers, ce sont les fameux **packages** !

Les fichiers nécessaires

Avant de créer notre premier package, nous allons créer un nouveau programme appelé `Test`, dans lequel nous allons pouvoir tester toutes les procédures, fonctions, types, variables que nous allons créer dans notre package. Voici la structure initiale de ce programme :

Code : Ada

```
with Integer_Array;
use Integer_Array;

procedure Test is
    T : T_Vecteur;
begin
end Test;
```

C'est quoi ce package `integer_array` et ce type `T_Vecteur` ?

Ce package `integer_array`, c'est justement le package que nous allons créer et le type `T_Vecteur` est un « `ARRAY OF Integer` » de longueur 10 (pour l'instant). Pour l'instant, si l'on tente de compiler ce programme, nous nous trouvons face à deux obstacles :

- Il n'y a aucune instruction entre `BEGIN` et `END` !
- Le package `Integer_Array` est introuvable (tout simplement parce qu'il est inexistant)

Nous allons donc devoir créer ce package. Comment faire ? Vous allez voir, c'est très simple : cliquez sur `File > New` deux fois pour obtenir deux nouvelles feuilles vierges.

Deux fois ? On voulait pas créer un seul package ? Y'a pas erreur ?

Non, non ! J'ai bien dit **deux** fois ! Vous comprendrez pourquoi ensuite. Sur l'une des feuilles, nous allons écrire le code suivant :

Code : Ada

```
WITH Ada.Integer_Text_IO, Ada.Text_IO;
USE Ada.Integer_Text_IO, Ada.Text_IO;

PACKAGE BODY Integer_Array IS
END Integer_Array;
```

Puis, dans le même répertoire que votre fichier `Test.adb`, enregistrer ce nouveau fichier sous le nom `Integer_Array.adb`.

Dans le second fichier vierge, écrivez :

Code : Ada

```
PACKAGE Integer_Array IS
END Integer_Array;
```

Enfin, toujours dans le même répertoire, enregistrer ce fichier sous le nom `Integer_Array.ads`. Et voilà ! Vous venez de créer votre premier package ! Bon, j'admettrai qu'il est un peu vide, mais nous allons remédier à ce défaut.

Le deuxième fichier DOIT avoir l'extension `.ads` et pas `.adb` ! Sinon, vos packages ne marcheront pas.

Notre première procédure... empaquetée

Vous remarquerez que cela ressemble un peu à ce que l'on a l'habitude d'écrire, sauf qu'au lieu de créer une procédure, on crée un package.

Ça me dit toujours pas pourquoi je dois créer deux fichiers qui comportent peu ou prou la même chose ?!

Nous y arrivons. Pour comprendre, nous allons créer une procédure qui affiche un tableau. Vous vous souvenez ? Nous en avons déjà écrit plusieurs (souvent appelées *Afficher*). Moi, je décide que je vais l'appeler `Put()` comme c'est d'usage en Ada.

Euh... il existe déjà des procédures `Put()` et elles ne sont pas faites pour les tableaux ! Ça sent le plantage ça, non ?

Encore une fois, rassurez-vous et faites moi confiance, tout va bien se passer et je vous expliquerai tout ça à la fin (« Oui, ça fait beaucoup de choses inexplicables », dira Mulder & Scully 😊). Dans le fichier `Test.adb`, nous allons écrire :

Code : Ada - test.adb

```
WITH Integer_Array;
USE Integer_Array;

PROCEDURE Test IS
    T : T_Vecteur(1..10);
BEGIN
    T := (others => 0);
    Put(T,2);
END Test;
```

Je voudrais, en effet, que notre procédure `Put()` puisse également gérer le nombre de chiffres affichés, si besoin est. Puis, dans le fichier `Integer_Array.ads`, nous allons ajouter ces lignes entre `IS` et `END` :

Code : Ada - Integer_Array.ads

```
TYPE T_Vecteur IS ARRAY(integer range <>) OF Integer;
PROCEDURE Put (T : IN T_Vecteur; Fore : IN Integer := 11);
```

Je prends les devants : il n'y a aucune erreur, j'ai bien écrit un point-virgule à la fin de la procédure et pas le mot `IS`. Pourquoi ? Pour la simple et bonne raison que notre fichier `Integer_Array.ads` n'est pas fait pour contenir des milliers de lignes de codes et d'instructions ! Il ne contient qu'une liste de tout ce qui existe dans notre package. On dit que la ligne « `PROCEDURE Put (T : IN T_Vecteur; Fore : IN Integer := 11);` » est la **spécification** (ou le **prototype**) de la procédure `Put()`. La spécification de notre procédure ne contient pas les instructions nécessaires à son fonctionnement, il indique seulement au compilateur : *avou là, j'ai une procédure qui s'appelle Put() et qui a besoin de ces 2 paramètres !*

Le contenu de notre procédure (ce que l'on appelle le **corps** ou **BODY** en Ada), nous allons l'écrire dans notre fichier

Integer_Array.adb, entre **IS** et **END**. Vous pouvez soit reprendre l'une de vos procédures d'affichage déjà créées et la modifier, soit recopier la procédure ci-dessous :

Code : Ada

```
PROCEDURE Put(T : IN T_Vecteur ; Fore : IN Integer := 11) IS
BEGIN
  FOR I IN T'RANGE LOOP
    Put(T(I),Fore);
    put(" ; ");
  END LOOP;
END Put;
```

Enregistrer vos fichiers, compiler votre programme *Test.adb* et lancez-le : vous obtenez une jolie suite de 0 séparés par des points virgules. Cela nous fait donc un total de trois fichiers, dont voici les codes :

- *Test.adb* :

Secret (cliquez pour afficher)

Code : Ada

```
WITH Integer_Array ;
USE Integer_Array ;

PROCEDURE Test IS
  T : T_Vecteur(1..10) ;
BEGIN
  T := (others => 0) ;
  Put(T,2) ;
END Test ;
```

- *Integer_Array.ads* :

Secret (cliquez pour afficher)

Code : Ada

```
PACKAGE Integer_Array IS
  TYPE T_Vecteur IS ARRAY(Integer range <>) OF Integer ;
  PROCEDURE Put (T : IN T_Vecteur; Fore : IN Integer := 11);
END Integer_Array ;
```

- *Integer_Array.adb* :

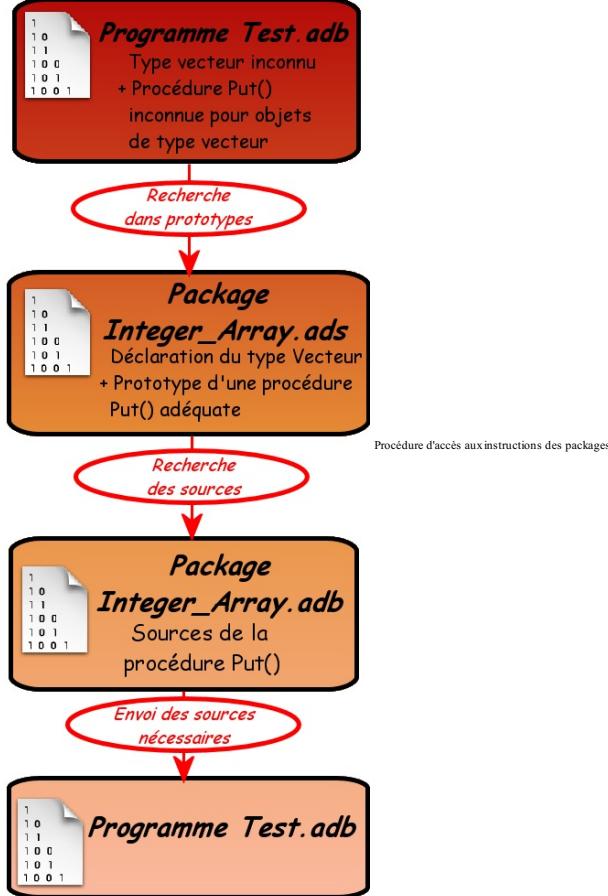
Secret (cliquez pour afficher)

Code : Ada

```
WITH Ada.Integer_Text_IO, Ada.Text_IO ;
USE Ada.Integer_Text_IO, Ada.Text_IO ;

PACKAGE BODY Integer_Array IS
  PROCEDURE Put(T : IN T_Vecteur ; Fore : IN Integer := 11)
  IS
  BEGIN
    FOR I IN T'RANGE LOOP
      Put(T(I),Fore) ;
      put(" ; ");
    END LOOP ;
  END Put ;
END Integer_Array ;
```

Nous pourrions résumer la situation par ce schéma :



Reste l'histoire de la procédure `Put()`. La procédure `Put()` ne constitue pas un mot réservé du langage, pour l'utiliser vous êtes obligés de faire appel à un package :

- `Integer_Text_IO` pour afficher un `Integer` ;
- `Float_Text_IO` pour afficher un `float` ;
- notre package `Integer_Array` pour afficher un `T_Vecteur`.

À chaque fois, c'est une procédure Put() différente qui apparaît et ce qui les différencie pour le compilateur, ce sont le nombre et le type des paramètres de ces procédures. Pour cela, GNAT s'appuie sur les spécifications qu'il trouve dans les fichiers.ads. En cas d'hésitation, le compilateur vous demandera de spécifier de quelle procédure vous parlez en écrivant :

- Integer_Text_IO.Put() ou
- Float_Text_IO.Put() ou
- Integer_Array.Put().

Variables et constantes globales

Je vous ai plusieurs fois dit qu'une variable ne pouvait exister en dehors de la procédure, de la fonction ou du bloc de déclaration où elle a été créée. Ainsi, si l'est impossible que deux variables portent le même nom dans un même bloc, il est toutefois possible que deux variables aient le même nom si elles sont déclarées dans des blocs distincts (par bloc, j'entends procédure, fonction ou bloc **DECLARE**). De même, je vous ai également expliqué que le seul moyen pour qu'un sous-programme et le programme principal puissent se transmettre des valeurs, c'est en utilisant les paramètres. Eh bien tout cela est faux.



Quoi ? Mais pourquoi m'avoir menti ?

Il ne s'agit pas d'un mensonge mais d'une imprécision. Tout ce que je vous ai dit est à la fois le cas général et la méthode qu'il est bon d'appliquer pour générer un code de qualité. Toutefois, il est possible de créer des variables qui outrepasse ces restrictions. Ce sont ce que l'on appelle les **variables globales**. Il s'agit de variables ou constantes déclarées dans un package. Dès lors, tout programme utilisant ce package y aura accès : elles seront créées au démarrage du programme et ne seront détruites qu'à sa toute fin. De plus, un sous-programme pourra y faire appel sans qu'elle ne soit citée dans les paramètres.

Vous vous demandez sûrement pourquoi nous n'avons pas commencé par là ? Vous devez savoir qu'utiliser des variables globales est très risqué et que c'est une pratique à proscrire autant que possible. Imaginons une fonction itérative (utilisant une boucle) qui modifieait notre variable dans le but de calculer son résultat. La fonction se termine, transmet son résultat et libère la mémoire qu'elle avait requisé : ses variables internes sont alors supprimées... mais pas la variable globale qui continue d'exister. Sauf que sa valeur a été modifiée par la fonction, et la disparition de cette dernière ne réinitialisera pas la variable globale. Qui sait alors combien d'itérations ont été effectuées ? Comment retrouver la valeur initiale pour rétablir un fonctionnement normal du programme ?

L'usage des variables globales doit donc être limité aux cas d'absolu nécessité. Si par exemple, une entreprise dispose d'un parc d'imprimantes en réseaux, quelques variables globales enregistrent le nombre total d'imprimantes ou le nombre d'imprimantes libres seront nécessaires. Et encore, le nombre d'imprimantes libres peut être modifié par plusieurs programmes successivement ou simultanément ! Cela crée de nombreux problèmes que nous découvrirons quand nous aborderons le **multitasking**.

L'usage le plus fréquent (et que je vous autorise volontiers) est celui des constantes globales. Celles-ci peuvent vous permettre d'enregistrer des valeurs que vous ne souhaitez pas réécrire à chaque fois afin d'éviter toute erreur. Il peut s'agir par exemple du titre de votre jeu, des dimensions de la fenêtre de votre programme ou de nombres spécifiques comme dans l'exemple ci-dessous :

Code : Ada - P_Constantes.ads

```
PACKAGE P_Constantes IS
  Pi      : Constant Float := 3.14159 ;          --Le célèbre
  nombre Pi
  Exp    : Constant Float := 2.71828 ;          --La constante
  de Neper
  Phi    : Constant Float := 1.61803 ;          --Le nombre
  d'or
  Avogadro : Constant Float := 6,022141 * 10.0**23 ; --La constante
  d'Avogadro
END P_Constantes ;
```

Et voilà quelques fonctions utilisant ces constantes :

Code : Ada

```
WITH P_Constantes ;      USE P_Constantes ;
...
FUNCTION Perimetre(R : Float) RETURN Float IS
BEGIN
  RETURN 2.0 * Pi * R ;
END Perimetre ;
...
FUNCTION Exponentielle(x : Float) RETURN Float IS
BEGIN
  RETURN Exp ** x ;
END Exponentielle ;
...
```

Vous remarquerez que Pi ou Exp n'ont jamais été déclarées dans mes fonctions ni même passées en paramètre. Elles jouent ici pleinement leur rôle de constante globale : il serait idiot de calculer le périmètre d'un cercle avec autre chose que le nombre π !



Tu ne crées pas de fichier.ads ?

C'est dans ce cas, complètement inutile : mon package ne comporte que des constantes. Je n'ai pas de corps à détailler. Les fichiers.adb (Ada Body), ne sont utiles que s'il faut décrire des procédures ou des fonctions. Si le package ne comporte que des types ou variables globales, le fichier.ads (Ada Specifications) suffit. Rappelez-vous le schéma de tout à l'heure : les programmes cherchent d'abord les spécifications dans le fichier.ads !

Trouver et classer les fichiers

Les packages fournis avec GNAT

Où se trouvent les packages Ada ?



Mais alors, si un package n'est rien de plus que deux fichiers.adb et.ads, il doit bien exister des fichiers *ada.text_io.ads*, *ada.text_io.adb*... non ?

En effet. Tous les packages que nous citons en en-tête depuis le début de ce cours ont des fichiers correspondants : un fichier.ads et éventuellement un fichier.adb. Pour découvrir des fonctionnalités supplémentaires de Ada.Text_IO, Ada.Numerics.Discrete_Random ou Ada.Strings.Unbounded, il suffit simplement de trouver ces fichiers.ads et des les examiner. Il est inutile d'ouvrir le corps des packages, en général peu importe la façon dont une procédure a été codée, ce qui compte c'est ce qu'elle fait. Il suffit donc d'ouvrir les spécifications, de connaître quelques mots d'Anglais et de lire les commentaires qui expliquent succinctement le rôle de tel ou tel sous-programme ainsi que le rôle des divers paramètres.

Pour trouver ces fichiers, ouvrez le répertoire d'installation de GNAT. Vous devriez tout d'abord trouver un répertoire correspondant à la version de votre compilateur (l'année en fait). Ouvrez-le puis suivez le chemin suivant (il est possible que les noms varient légèrement selon votre système d'exploitation ou la version de votre compilateur) :

`lib\gcc\i686-pc-mingw32\4.5.3\adainclude`

Convention de nommage



Mais ... c'est un vrai chantier ! Il y a des centaines de fichier et les noms sont incompréhensibles !

C'est vrai que c'est déconcertant au début, mais vous parviendrez peu à peu à vous retrouver dans cette énorme pile de fichiers. Commencez déjà par les classer par type afin de séparer le corps des spécifications. Vous devriez remarquer que, à quelques rares exceptions, tous les noms de fichier commencent par une lettre suivie d'un tiret. Cette lettre est l'initiale du premier mot du nom de package. Par exemple le package Ada.Text_IO a un nom de fichier commençant par « a- ». La convention est la suivante :

Lettre	Nom complet	Remarque
a-	Ada	Il s'agit des principaux packages auxquels vous ferez appel.
g-	Gnat	Nous ne ferons que très rarement appel à ces packages.
i-	Interfaces	Ces packages servent à faire interagir les codes en Ada avec des codes en C, en Fortran ...
s-	System	Nous n'utiliserons jamais ces packages.

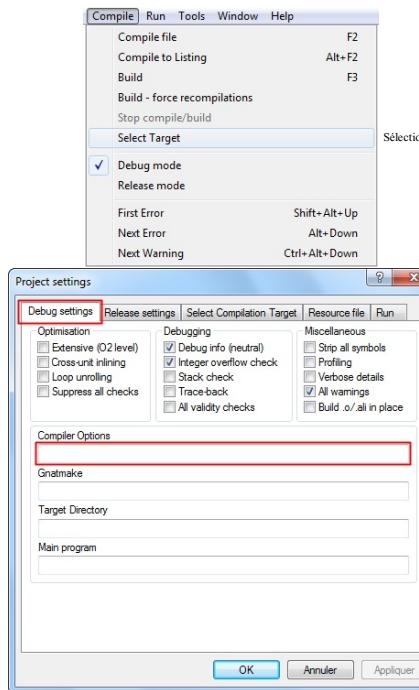
Vous remarquerez ensuite que les points dans Ada.Text_IO ou Ada.Strings.Unbounded sont remplacés par des tirets dans les noms de fichier. Ainsi Ada.Text_IO a un fichier nommé *a-textio.ads*, Ada.Strings.Unbounded a un fichier nommé *a-strnib.ads*.

Organiser nos packages



Ne serait-il pas possible de classer nos packages dans des répertoires nous aussi ?

Bien sûr que si. Vous verrez que lorsque vos programmes prendront de l'ampleur, le nombre de packages augmentera lui-aussi. Il deviendra donc important de classer tous ces fichiers pour mieux vous y retrouver. Si le logiciel GPS gère seul l'ensemble du projet, l'IDE Adagide est lui plus rudimentaire : placez par exemple vos packages dans un sous-répertoire nommé « *packages* » (original non ?). Bien sûr, si vous cherchez à compiler, Adagide vous dira qu'il ne trouve pas vos packages. Il faut donc lui indiquer où ils se situent. Cliquez sur menu *Compile* > *Select Target*. Dans la fenêtre qui s'ouvre, choisissez l'onglet *Debug settings* où vous devriez trouver la ligne *Compiler Options*. C'est là qu'il vous faudra inscrire : `-I ./packages`. Cette ligne signifie « *Inclure (-I) le répertoire ./packages* ».



Compiler et construisez votre programme : ça marche !



GNAT devrait créer un fichier appelé *gnat.ago*. Il sera désormais interdit de le supprimer car c'est lui qui enregistre vos options et donc l'adresse où se situent vos packages.

Petite astuce pour limiter les fichiers générés

Vous avez du remarquer qu'en compilant votre code ou en construisant votre programme, GNAT génère toute une flopée de fichiers aux extensions étranges : `.o`, `.ali`, `.bak`, `.hk1`, `.ago` ... sans parler des copies de vos fichiers sources commençant par `b_` ! Et en éitant des packages, nous augmentons encore le nombre des fichiers qui seront générés ! Ces fichiers sont utiles pour GNAT et permettent également d'accélérer la compilation ; mais lorsqu'ils sont trop nombreux, cela devient une perte de temps pour vous. Voici une astuce pour faire le ménage rapidement.

Ouvrez un éditeur de texte et copiez-y les commandes suivantes :

```
Code : autre
del *.o
del *.ali
del *.bk.*
del *.hk1
del *.0
del *.1
del *.2
del *.3
del *.4
del *.5
del *.6
del *.7
del *.8
del b~*
```

Enregistrez ensuite ce fichier texte dans le répertoire de votre programme en lui donnant le nom *Commande.bat*. Désormais, il suffira que vous double-cliquez dessus pour qu'un grand ménage soit fait, supprimant tous les fichiers dont l'extension est notée ci-dessus. Remarquez d'ailleurs qu'il n'y a pas d'instruction `del gnat.ag0` afin de conserver ce fichier.

Compléter notre package (exercices)

Cahier des charges

Pour l'heure, notre package ne contient pas grand chose : le type `T_Vecteur` et une procédure `Put()`. Nous allons donc le compléter, ce sera un excellent exercice pour appliquer tout ce que nous avons vu et cela nous fournira un package bien utile. De quoi avons-nous besoin ?

- Des procédures d'affichage;
- Des procédures de création;
- Des fonctions de tri;
- Des fonctions pour les opérations élémentaires.

Procédures d'affichage

Code : Ada

```
PROCEDURE Put(T : IN T_Vecteur ; FORE : IN integer := 11) ;
```

Celle-là ne sera pas compliquée : nous l'avons déjà faite. Elle affiche les valeurs contenues dans un `T_Vecteur` T sur une même ligne. `Fore` indique le nombre de chiffres à afficher (`Fore` pour `Before` = Avant la virgule).

Code : Ada

```
PROCEDURE Put_line(T : IN T_Vecteur ; FORE : IN integer := 11) ;
```

Affiche les valeurs du `T_Vecteur` T comme `Put()` et retourne à la ligne.

Code : Ada

```
PROCEDURE Put_Column(T : IN T_Vecteur ; FORE : IN integer := 11 ;
Index : IN Boolean := true) ;
```

Affiche les valeurs du T_Vecteur T comme Put() mais en retournant à la ligne après chaque affichage partiel. Le paramètre booléen Index indique si l'on souhaite afficher l'indice de la valeur affichée (le numéro de sa case).

Procédures de création

Code : Ada

```
PROCEDURE Get(T : IN OUT T_Vecteur) ;
```

Permet à l'utilisateur de saisir au clavier les valeurs du T_Vecteur T. L'affichage devra être le plus sobre possible. Pas de « Veuillez vous donner la peine de saisir la 1ere valeur du T_Vecteur en question » :

Code : Ada

```
PROCEDURE Init(T : IN OUT T_Vecteur ; Val : IN integer := 0) ;
```

Initialise un tableau en le remplaçant de 0 par défaut. Le paramètre Val permettra de compléter par autre chose que des 0.

Code : Ada

```
PROCEDURE Generate(T : IN OUT T_Vecteur ;
Min : integer := integer'first ;
Max : integer := integer'last) ;
```

Génère un T_Vecteur rempli de valeurs aléatoires comprises entre Min et Max. Il vous faudra ajouter le package Ada.Numerics.Discrete_Random au début de votre fichier.adb !

Fonctions de tri

Code : Ada

```
FUNCTION Tri_Selection(T : T_Vecteur) RETURN T_Vecteur ;
```

Pour l'instant, nous n'avons vu qu'un seul algorithme de tri de tableau : le tri par sélection (ou tri par extraction). Je vous invite donc à en faire un petit copier-coller-modifier. Mais il en existe de nombreux autres. Nous serons donc amenés à créer plusieurs fonctions de tri, d'où le nom un peu long de notre fonction. J'ai préféré le terme français cette fois (Select_Sort n'étant pas très parlant). Peut-être sera-t-il intéressant de créer également les fonctions RangMin ou Echanger pour un usage futur ?

Opérations élémentaires



Les opérations ci-dessous seront détaillées dans la sous-partie « Vecteurs et calcul vectoriel ».

Code : Ada

```
FUNCTION Somme(T : T_Vecteur) RETURN integer ;
```

Renvoie la somme des éléments du T_Vecteur.

Code : Ada

```
FUNCTION "+"(Left,Right : T_Vecteur) RETURN T_Vecteur ;
```

Effectue la somme de deux T_Vecteurs. L'écriture très spécifique de cette fonction (avec un nom de fonction écrit comme un string et exactement deux paramètres Left et Right pour Gauche et Droite) nous permettra d'écrire par la suite T3 := T1 + T2 au lieu de T3 := +(T1,T2) ! On dit que l'on surcharge l'opérateur "+".

Code : Ada

```
FUNCTION "*"*(Left : integer ; Right : T_Vecteur) RETURN T_Vecteur ;
```

Effectue le produit d'un nombre entier par un T_Vecteur.

Code : Ada

```
FUNCTION "*"*(Left, Right : T_Vecteur) RETURN Integer ;
```

Effectue le produit scalaire de deux T_Vecteurs. Pas de confusion possible avec la fonction précédente car les paramètres ne sont pas de même type.

Code : Ada

```
FUNCTION Minimum(T : T_Vecteur) RETURN Integer ;
FUNCTION Maximum(T : T_Vecteur) RETURN Integer ;
```

Renvoient respectivement le plus petit et le plus grand élément d'un T_Vecteur.

Solutions

Voici les packages Integer_Array.adb et Integer_Array.ads tels que je les ai écrits. Je vous conseille d'essayer de les écrire par vous-même avant de copier-coller ma solution, ce sera un très bon entraînement 😊 Autre conseil, pour les opérations élémentaires, je vous invite à lire la sous-partie qui suit afin de comprendre les quelques ressorts mathématiques.

- Integer_Array.adb :

Code : Ada - Integer_Array.adb

```
WITH Ada.Integer_Text_IO, Ada.Text_IO,
Ada.Numerics.Discrete_Random ;
USE Ada.Integer_Text_IO, Ada.Text_IO ;

package body Integer_Array IS

    -- Procédures d'affichage --
    -- -----
    PROCEDURE Put (
        T : IN      T_Vecteur;
        Fore : IN      Integer := 11) IS
    BEGIN
        FOR I IN T'RANGE LOOP
            Put(T(I),Fore) ;
            put(" ; ") ;
        END LOOP ;
    END Put ;

    PROCEDURE Put_Column (
        T : IN      T_Vecteur;
        Fore : IN      Integer := 11) IS
```

```

BEGIN
  FOR I IN T'RANGE LOOP
    Put(T(I),Fore) ;
    new_line ;
  END LOOP ;
END Put_Column ;

PROCEDURE Put_Line (
  T   : IN      T_Vecteur;
  Fore : IN      Integer := 11) IS
BEGIN
  Put(T,Fore) ;
  New_Line ;
END Put_Line ;

-----Procedures de saisie-----

PROCEDURE Init (
  T   : OUT T_Vecteur;
  Value : IN      Integer := 0) IS
BEGIN
  T := (OTHERS => Value) ;
END Init ;

procedure generate(T : in out T_Vecteur;
  min : in integer := integer'first;
  max : in integer := integer'last) is
  subtype Random_Range is integer range min..max;
  Package Random_Array is
    use Ada.Numerics.Discrete_Random(Random_Range);
    G : generator;
begin
  reset(G);
  T := (others => random(g));
end generate;

procedure get(T : in out T_Vecteur) is
begin
  for i in T'range loop
    put(integer'image(i) & " : ");
    get(T(i));
    skip_line;
  end loop;
end get;

-----Fonctions de tri-----

function Tri_Selection(T : T_Vecteur) return T_Vecteur is
  function RangMin(T : T_Vecteur ; debut : integer ; fin :
integer) return integer is
    Rang : integer := debut;
    Min : integer := T(debut);
  begin
    for i in debut..fin loop
      if T(i)<Min
        then Min := T(i);
        Rang := i;
      end if;
    end loop;
    return Rang;
  end RangMin;

  procedure Echanger(a : in out integer ; b : in out
integer) is
    c : integer;
  begin
    c := a;
    a := b;
    b := c;
  end Echanger;

  Tab : T_Vecteur := T;
begin
  for i in Tab'range loop
    Echanger(Tab(i),Tab(RangMin(Tab,i,Tab'last)));
  end loop;
  return Tab;
end Tri_Selection;

-----Opérations élémentaires-----

function Somme(T:T_Vecteur) return integer is
  S : integer := 0;
begin
  for i in T'range loop
    S := S + T(i);
  end loop;
  return S;
end somme;

FUNCTION "+" (Left,Right : T_Vecteur) return T_Vecteur is
  T : T_Vecteur;
begin
  for i in Left'range loop
    T(i) := Left(i) + Right(i);
  end loop;
  return T;
end "+";

FUNCTION "*" (Left : integer ; Right : T_Vecteur) return
T_Vecteur is
  T : T_Vecteur;
begin
  for i in Right'range loop
    T(i) := Left * Right(i);
  end loop;
  return T;
end "*";

FUNCTION "***" (Left,Right : T_Vecteur) return Integer is
  S : Integer := 0;
begin
  for i in Left'range loop
    S := S + Left(i) * Right(i);
  end loop;
  return S;
end "***;

FUNCTION Minimum(T : T_Vecteur) return integer is
  min : integer := T(T'first);
BEGIN
  FOR I in T'range loop
    if T(i)<min
      then min := T(i);
    end if;
  end loop;
  return min;
end Minimum;

FUNCTION Maximum(T : T_Vecteur) return integer is
  max : integer := T(T'first);
BEGIN
  FOR I in T'range loop
    if T(i)>max
      then max := T(i);
    end if;
  end loop;
  return max;
end Maximum;

END Integer_Array;

```

- Integer_Array.ads :
Code : Ada - Integer_Array.ads

```
PACKAGE Integer_Array IS
```

```

-----  

--Définition des types et variables--  

Taille : CONSTANT Integer := 10;  

TYPE T_Vecteur IS ARRAY(1..Taille) OF Integer ;  

--Procédures d'affichage--  

PROCEDURE Put (  

    T : IN      T_Vecteur;  

    Fore : IN     Integer := 11);  

PROCEDURE Put_Column (  

    T : IN      T_Vecteur;  

    Fore : IN     Integer := 11);  

PROCEDURE Put_Line (  

    T : IN      T_Vecteur;  

    Fore : IN     Integer := 11);  

--Procédures de saisie--  

PROCEDURE Init (  

    T : OUT T_Vecteur;  

    Value : IN     Integer := 0);  

PROCEDURE Generate (  

    T : IN OUT T_Vecteur;  

    Min : IN     Integer := Integer'First;  

    Max : IN     Integer := Integer'Last);  

PROCEDURE Get (  

    T : IN OUT T_Vecteur);  

--Fonctions de tri--  

function Tri_Selection(T : T_Vecteur) return T_Vecteur ;  

--Opérations élémentaires--  

function Somme(T:T_Vecteur) return integer ;  

FUNCTION "+" (Left,Right : T_Vecteur) return T_Vecteur ;  

FUNCTION "*" (Left : integer ; Right : T_Vecteur) return T_Vecteur ;  

FUNCTION "***" (Left,Right : T_Vecteur) return Integer ;  

FUNCTION Minimum(T : T_Vecteur) return integer ;  

FUNCTION Maximum(T : T_Vecteur) return integer ;  

END Integer_Array ;

```

Vous remarquerez que j'utilise autant que possible les attributs et agrégats. Pensez que vous serez amenés à réutiliser ce package régulièrement et sûrement aussi à modifier la longueur de nos tableaux T_Vecteur. Il ne faut pas que vous ayez besoin de refaire toutes vos procédures et fonctions pour les modifier !

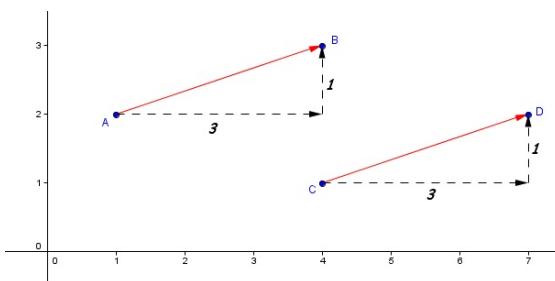
Vecteurs et calcul vectoriel (optionnel)

Qu'est-ce exactement qu'un T_Vecteur ?

 Depuis le début de ce chapitre, nous parlons de T_Vecteur au lieu de tableau à une dimension. Pourquoi ?

Le terme T_Vecteur (ou Vector) n'est pas utilisé par le langage Ada, contrairement à d'autres. Pour Ada, cela reste toujours un **ARRAY**. Mais des analogies peuvent être faites avec l'objet mathématique appelé « vecteur ». Alors, sans faire un cours de Mathématiques, qu'est-ce qu'un vecteur ?

De manière (très) schématique un T_Vecteur est la représentation d'un déplacement, symbolisé par une flèche :



Représentation de deux vecteurs égaux

Ainsi, sur le schéma ci-dessus, le T_Vecteur \vec{AB} se déplace de 3 unités vers la droite et de 1 unité vers le haut. Il représente donc le même déplacement que le T_Vecteur \vec{CD} et on peut écrire $\vec{AB} = \vec{CD}$. On remarquera que ces vecteurs sont « parallèles », de même « longueur » et dans le même sens. Du coup, on peut dire que ces deux vecteurs ont comme coordonnées : $(3; 1)$ ou $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$ ce qui devrait commencer à vous rappeler nos tableaux, non ?  Pour les récalcitrants, sachez qu'il est possible en Mathématiques de travailler avec des vecteurs en 3, 4, 5... dimensions. Les coordonnées d'un vecteur ressemblent alors à cela :

$$(3; 1; 5; 7; 9) \text{ ou } \begin{pmatrix} 3 \\ 1 \\ 5 \\ 7 \\ 9 \end{pmatrix}$$

Ça ressemble sacrément à nos tableaux T_Vecteurs, non ? 

Calcul vectoriel

Plusieurs opérations sont possibles sur les T_Vecteurs. Tout d'abord, l'addition ou Relation de Chasles. Sur le schéma ci-dessus, chacun des vecteurs \vec{AB} ou \vec{CD} a été décomposé en deux vecteurs noirs tracés en pointillés : que l'on file de A à B en suivant le vecteur \vec{AB} ou en suivant le premier vecteur noir puis le second, le résultat est le même, on arrive en B ! Eh bien, suivre un vecteur puis un autre, cela revient à les additionner.

D'où la formule :

$$\begin{pmatrix} 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

On remarque qu'il a suffi d'ajouter chaque coordonnée avec celle « en face ». Il est aussi possible de multiplier un vecteur par un nombre. Cela revient simplement à l'allonger ou à le rétrécir (notez que multipliez un vecteur par un nombre négatif changera le sens du vecteur). Par exemple :

$$5 \times \begin{pmatrix} 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 15 \\ 5 \end{pmatrix}$$

Troisième opération : le [produit scalaire](#). Il s'agit ici de multiplier deux vecteurs entre eux. Nous ne rentrerons pas dans le détail, ce serait trop long à expliquer et sans rapport avec notre sujet. Voici comment effectuer le produit scalaire de deux vecteurs :

$$\begin{pmatrix} 3 \\ 1 \\ 5 \end{pmatrix} \times \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix} = (3 \times 2) + (1 \times 4) + (5 \times 6) = 6 + 4 + 30 = 40$$



Si le produit scalaire de deux vecteurs vaut 0, c'est qu'ils sont «perpendiculaires».

Bien, nous en avons fini avec l'explication mathématique des vecteurs. Pour ceux qui ne connaissaient pas les vecteurs, ceci devrait avoir quelque peu éclairé votre lanterne. Pour ceux qui les connaissaient déjà, vous me pardonnerez d'avoir pris quelques légèretés avec la rigueur mathématique pour pouvoir « vulgariser » ce domaine fondamental des Maths.

Nous avons maintenant un joli package Integer_Array. Nous pourrons l'utiliser aussi souvent que possible, dès lors que nous aurons besoin de tableaux à une seule dimension contenant des entiers. Il vous est possible également de créer un package Integer_Array2D pour les tableaux à deux dimensions. Mais surtout, conserver ce package car nous le compléterons au fur et à mesure avec de nouvelles fonctions de tri, plus puissantes et efficaces, notamment lorsque nous aborderons la récursivité ou les notions de complexité d'algorithme. Lors du chapitre sur la généricité, nous ferons en sorte que ce package puisse être utilisé avec des float, des natural... Bref, nous n'avons pas fini de l'améliorer.

Qui plus est, vous avez sûrement remarqué que ce chapitre s'appelle «La programmation modulaire I». C'est donc que nous aborderons plus tard un chapitre «La programmation modulaire II». Mais assurez-vous, vous en savez suffisamment aujourd'hui pour créer vos packages. Le chapitre II permettra d'améliorer nos packages et d'aborder les notions d'héritage et d'encapsulation chères à la programmation orientée objet.

En résumé :

- Un package est constitué d'un fichier ads contenant les spécifications de vos fonctions et procédures et, si nécessaire, d'un fichier adb contenant le corps. Ce dernier se distingue par l'utilisation des mots clés **PACKAGE BODY**.
- Prenez soin de vérifier que les corps des fonctions et procédures correspondent trait pour trait à leurs spécifications car ce sont ces dernières qui feront foi.
- Les types, les variables et constantes globales sont déclarés avec les spécifications des fonctions et procédures dans le fichier ads.
- Évitez d'utiliser des variables globales autant que cela est possible. En revanche, il n'est pas idiot de « globaliser » vos constantes.
- N'hésitez pas à fouiller parmi les packages officiels. Vous y trouverez des pépites qui vous éviteront de réinventer la roue ou vous ouvriront de nouvelles possibilités.

Les fichiers

Nos programmes actuels souffrent d'un grave inconvénient : tout le travail que nous pouvons effectué, tous les résultats obtenus, toutes les variables enregistrées en mémoire vive... ont une durée qui se limite à la durée d'utilisation de notre programme. Si l'on ferme, nous perdons toutes ces données, alors qu'elles pourraient s'avérer fort utiles pour une future utilisation.

Il sera donc utile de pouvoir les enregistrer, non pas en mémoire vive, mais sur le disque dur. Et pour cela, nous devons être capable de créer des fichiers, de les lire ou de les modifier. C'est d'ailleurs ainsi que procède la plupart des logiciels. Word ou OpenOffice ont besoin d'enregistrer le travail effectué par l'utilisateur dans un fichier .doc, .odt... De même, n'importe quel jeu a besoin de créer des fichiers de sauvegarde mais aussi de lire continuellement des fichiers (les images ou les vidéos notamment) s'il veut fonctionner correctement.

C'est donc aux fichiers que nous allons nous intéresser dans ce chapitre. Vous allez voir, cela ne sera pas très compliqué, car vous disposez déjà de la plupart des notions nécessaires. Il y a juste quelques points à comprendre avant de pouvoir nous amuser. ☺

Ouvrir / Fermer un fichier texte

Nous allons voir plusieurs types de fichiers utilisables en Ada. Le premier type est le fichier texte. La première chose à faire lorsque l'on veut manipuler un fichier, c'est de l'ouvrir ! ☺ Cette remarque peut paraître idiote et pourtant : toute modification d'un fichier existant ne peut se faire que si votre programme a préalablement ouvert le fichier concerné. Et bien entendu, la dernière chose à faire est de le fermer.

Package nécessaire

Le package à utiliser, vous le connaissez déjà : Ada.Text_IO ! Le même que nous utilisions pour afficher du texte ou en saisir sur l'écran de la console. En fait, la console peut être considérée comme une sorte de fichier qui s'ouvre et se ferme automatiquement.

Le type de l'objet

Comme toujours, nous devons déclarer nos variables ou nos objets avant de pouvoir les manipuler. Nous allons créer une variable composite de type fichier qui fera le lien avec le vrai fichier texte :

Code : Ada

```
MonFichier : File_Type ;
```

Fermer un fichier

Pour fermer un fichier, rien de plus simple, il suffit de taper l'instruction suivante (toujours entre BEGIN et END, bien sûr) :

Code : Ada

```
close(MonFichier) ;
```



En effet ! ☺ Si vous fermez un fichier qui n'est pas ouvert, alors vous aurez droit à cette magnifique erreur :

Code : Console

```
raised ADA.IO_EXCEPTIONS.STATUS_ERROR : file not open
```

Il faut donc ouvrir avant de fermer ! Mais je commence par la fermeture car c'est ce qu'il y a de plus simple.

Ouvrir un fichier

Il y a une subtilité lorsque vous désirez ouvrir un fichier, je vous demande donc de lire toute cette partie et la suivante en me faisant confiance. Pas de questions, j'y répondrai par la suite. Il faut distinguer deux cas pour ouvrir un fichier :

- Soit ce fichier existe déjà.
- Soit ce fichier n'existe pas encore.

Ouvrir un fichier existant

Voici la spécification de l'instruction permettant d'ouvrir un fichier existant :

Code : Ada

```
procedure open(File : in out file_type ;
              Mode : in File_Mode := Out_File ;
              Name : in string) ;
```

File est le nom de la variable fichier : ici, nous l'avons appelé MonFichier. Nous reviendrons dans la partie suivante sur Mode, mais pour l'instant nous allons ignorer vu qu'il est déjà prédefini. Name est le nom du fichier concerné, extension comprise. Par exemple, si notre fichier a l'extension .txt et s'appelle enregistrement, alors Name doit valoir "enregistrement.txt". Nous pouvons ainsi taper le code suivant :

Code : Ada

```
WITH Ada.Text_IO ;
USE Ada.Text_IO ;

PROCEDURE TestFichier IS
    MonFichier : File_Type ;
BEGIN
    open(MonFichier,Name => "enregistrement.txt") ;
    close(MonFichier) ;
END TestFichier ;
```

Seul souci, si le compilateur ne voit aucune erreur, la console nous affiche toutefois ceci :

Code : Console

```
raised ADA.IO_EXCEPTIONS.NAME_ERROR : enregistrement.txt: No error
```

Le fichier demandé n'existe pas. Deux possibilités :

- Soit vous le créez «à la main» dans le même répertoire que votre programme.
- Soit vous lisez la sous-sous-partie suivante. ☺

Créer un fichier

Voici la spécification de l'instruction permettant la création d'un fichier :

Code : Ada

```
procedure Create(File : in out File_Type;
                 Mode : in File_Mode := Out_File;
                 Name : in String);
```

Elle ressemble beaucoup à la procédure open(). Et pour cause, elle fait la même chose (ouvrir un fichier) à la différence qu'elle

commence par créer le fichier demandé. Nous pouvons donc écrire le code suivant :

Code : Ada

```
WITH Ada.Text_IO;
USE Ada.Text_IO;

PROCEDURE TestFichier IS
    MonFichier : File_type;
BEGIN
    create(MonFichier, Name => "enregistrement.txt");
    close(MonFichier);
END TestFichier;
```

Et cette fois pas d'erreur après compilation ! Ça marche !



Euh... ça marche, ça marche... c'est vite dit. Il ne se passe absolument rien ! Je pensais que mon programme allait ouvrir le fichier texte et que je le verrais à l'écran ?!



Non ! Ouvrir un fichier permet seulement au logiciel de le lire ou de le modifier (au **logiciel** seulement). Pour que l'utilisateur puisse voir le fichier, il faut qu'il ouvre un logiciel (le bloc-note par exemple) qui ouvrira à son tour et affichera le fichier demandé.

Dans l'exemple donné ci-dessus, notre programme ouvre un fichier, et ne fait rien de plus que de le fermer. Pas d'affichage, pas de modification... Nous verrons dans ce chapitre comment modifier le fichier, mais nous ne verrons que beaucoup plus tard comment faire un bel affichage à la manière des éditeurs de texte (dernière partie du cours).



Alors deuxième question : que se passe-t-il si le fichier existe déjà ?

Eh bien testez ! C'est la meilleure façon de le savoir. Si vous avez déjà lancé votre programme, alors vous disposez d'un fichier "enregistrement.txt". Ouvrez-le avec NotePad par exemple et tapez quelques mots dans ce fichier. Enregistrez, fermez et relancez votre programme. Ouvrez à nouveau votre fichier : il est vide !

La procédure open() se contente d'ouvrir le fichier, mais celui-ci doit déjà exister. La procédure create() crée le fichier, quoiqu'il arrive. Si un fichier porte déjà le même nom, il est supprimé et remplacé par un nouveau, vide.

Détruire un fichier



Puisqu'il est possible de créer un fichier, il doit être possible de le supprimer ?

Tout à fait. Voici le prototype de la procédure en question :

Code : Ada

```
procedure Delete (File : in out File_Type);
```



Attention ! Pour supprimer un fichier, vous devez auparavant l'avoir ouvert !

Le paramètre Mode

Nous avons laissé une difficulté de côté tout à l'heure : le paramètre Mode. Il existe trois modes possibles pour les fichiers en Ada (et dans la plupart des langages) :

- Lecture seule : In_File
- Écriture seule : Out_File
- Ajout : Append_File



Ça veut dire quoi ?

Lecture seule

En mode Lecture seule, c'est-à-dire si vous écrivez ceci :

Code : Ada

```
Open(MonFichier, In_File, "enregistrement.txt");
```

Vous ne pourrez que lire ce qui est écrit dans le fichier, impossible de le modifier. C'est un peu comme si notre fichier était un paramètre en mode IN.

Écriture seule

En mode Écriture seule, c'est-à-dire si vous écrivez ceci :

Code : Ada

```
Open(MonFichier, Out_File, "enregistrement.txt");
```

Vous ne pourrez qu'écrire dans le fichier à partir du début, en écrasant éventuellement ce qui est déjà écrit, impossible de lire ce qui existe déjà. C'est un peu comme si notre fichier était un paramètre en mode OUT.

Ajout

Le mode Ajout, c'est-à-dire si vous écrivez ceci :

Code : Ada

```
Open(MonFichier, Append_File, "enregistrement.txt");
```

Vous pourrez écrire dans le fichier, comme en mode Out_File. La différence, c'est que vous n'écrivez pas en partant du début, mais de la fin. Vous ajouterez des informations à la fin au lieu de supprimer celles existantes.



Et il n'y aurait pas un mode IN OUT des fois ?

Eh bien non. Il va falloir faire avec. D'ailleurs, même Word ne peut guère faire mieux : soit il écrit dans le fichier au moment de la sauvegarde, soit il le lit au moment de l'ouverture. Les autres opérations (mise en gras, changement de police, texte nouveau...) ne sont pas effectuées directement dans le fichier mais seulement en mémoire vive et ne seront inscrites dans le fichier que lorsqu'Word les enregistrent.

Opérations sur les fichiers textes

Une fois notre fichier ouvert, il serait bon de pouvoir effectuer quelques opérations avec. Attention, les opérations disponibles ne seront pas les mêmes selon que vous avez ouvert votre fichier en mode In_File ou en mode Out_File/Append_File !

Mode lecture seule : In_File

Saisir un caractère

Tout d'abord vous allez devoir tout au long de ce chapitre imaginer votre fichier et l'emplacement du curseur (voir la figure suivante):



Dans l'exemple précédent, vous pouvez vous rendre compte que mon curseur se trouve à la seconde ligne, vers la fin du mot «*situé*». Ainsi, si je veux connaître le prochain caractère, je vais devoir utiliser la procédure `get()` que vous connaissez déjà, mais de manière un peu différente :

Code : Ada

```
procedure Get (File : in File_Type; Item : out Character);
```

Comme vous pouvez vous en rendre compte en lisant ce prototype, il ne suffira pas d'indiquer entre les parenthèses dans quelle variable de type caractère vous voulez enregistrer le caractère saisi, il faudra aussi indiquer dans quel fichier ! Comme dans l'exemple ci-dessous :

Code : Ada

```
Get(MonFichier,C);
```



Une fois cette procédure utilisée, la variable C vaudra 'é' (il y aura sûrement un problème avec le caractère latin) et le curseur sera déplacé à la fin du mot «*situé*», c'est-à-dire entre le 'é' et l'espace vide ' ' !

Autre remarque d'importance : si vous êtes en fin de ligne, en fin de page ou en fin de fichier, vous aurez droit à un joli message d'erreur ! Le curseur ne retourne pas automatiquement à la ligne. Pour pallier à cela, deux possibilités. Première possibilité : vous pouvez utiliser préalablement les fonctions suivantes :

Code : Ada

```
function End_Of_Line (File : in File_Type) return Boolean;
function End_Of_Page (File : in File_Type) return Boolean;
function End_Of_File (File : in File_Type) return Boolean;
```

Elle renvoie `TRUE` si vous êtes, respectivement, en fin de ligne, en fin de page ou en fin de fichier. Selon les cas, vous devrez alors utiliser ensuite les procédures :

Code : Ada

```
procedure Skip_Line (File : in File_Type; Spacing : in Positive_Count := 1);
procedure Skip_Page (File : in File_Type);
procedure Reset (File : in out File_Type);
```

`Skip_Line(MonFichier)` vous permettra de passer à la ligne suivante (nous l'utilisons jusque là pour vider le tampon). Il est même possible de lui demander de sauter plusieurs lignes grâce au paramètre `Spacing` en écrivant `Skip_Line(MonFichier,5)`. `Skip_Page(MonFichier)` vous permettra de passer à la page suivante. `Reset(MonFichier)` aura pour effet de remettre votre curseur au tout début du fichier. D'où le code suivant :

Code : Ada

```
loop
  if end_of_line(MonFichier)
    then skip_line(MonFichier);
  elsif end_of_page(MonFichier)
    then skip_page(MonFichier);
  elsif end_of_file(MonFichier)
    then reset(MonFichier);
    else get(C); exit;
  end if;
end loop;
```

Cette boucle permet de traiter les différents cas possibles et de recommencer tant que l'on n'a pas saisi un vrai caractère ! Attention, si le fichier est vide, vous risquez d'attendre un moment 🕒. Passons à la deuxième possibilité :

Code : Ada

```
procedure Look_Ahead(File : in File_Type;
                      Item : out Character;
                      End_Of_Line : out Boolean);
```

Cette procédure ne fait pas avancer le curseur mais regarde ce qui se trouve après : si l'on se trouve en fin de ligne, de page ou de fichier, la variable `End_Of_Line` vaudra true et le paramètre `Item` ne vaudra rien du tout. Dans le cas contraire, `End_Of_Line` vaudra false et `Item` aura la valeur du caractère suivant. Je le répète, le curseur n'est pas avancé dans ce cas là ! Voici un exemple avec une variable fin de type boolean et une variable C de type Character.

Code : Ada

```
Look_Ahead(MonFichier,C,fin);
if not fin
  then put("Le caractère vaut :"); put(C);
  else put("C'est la fin des haricots !");
end if;
```

Saisir un string

Pour saisir directement une ligne et pas un seul caractère, vous pourrez utiliser l'une des procédures dont voici les prototypes :

Code : Ada

```
procedure Get (File : in File_Type; Item : out String);
procedure Get_Line(File : in File_Type; Item : out String; Last : out Natural);
function Get_Line(File : in File_Type) return String;
```

La procédure `Get()` saisit tout le texte présent dans l'objet-fichier `File` et l'enregistre dans le string `Item` (le curseur se trouve alors à la fin du fichier). La procédure `get_line()` saisit tout le texte situé entre le curseur et la fin de la ligne et l'affecte dans le string `Item`; elle renvoie également la longueur de la chaîne de caractère dans la variable `Last` (le curseur sera alors placé au début de la ligne suivante). Enfin, la fonction `Get_Line` se contente de renvoyer le texte entre le curseur et la fin de ligne (comme la procédure, sauf l'utilisation et l'absence de paramètre `Last` différent).

Mais nous nous retrouvons face à un inconvénient de taille : nous ne connaissons pas à l'avance la longueur des lignes et donc la longueur des strings qui seront renvoyés. À cela, deux solutions : soit nous utilisons des strings suffisamment grands en prenant pour que les lignes ne soient pas plus longues (et il faudra prendre quelques précautions à l'affichage), soit nous faisons appel aux `unbounded_strings` :

- Première méthode :

Code : Ada

```
...
S : string(1..100);
MonFichier : file type;
```

```

    long : natural;
BEGIN
  open(MonFichier, In_File, "enregistrement.txt");
  get_line(MonFichier, S, long);
  put_line(S(1..long));
  ...

```

- Seconde méthode:

Code : Ada

```

    ...
  txt : unbounded_string;
  MonFichier : file_type;
BEGIN
  open(MonFichier, In_File, "enregistrement.txt");
  txt:=To_Unbounded_String(get_line(MonFichier));
  put_line(To_String(txt));
  ...

```

Mode écriture : Out_File / Append_File



Rappel : en mode Append_File, toute modification se fera à la fin du fichier ! En mode Out_File, les modifications se feront à partir du début, au besoin en écrasant ce qui était déjà écrit.

Vous devriez commencer à vous douter des opérations d'écriture applicables aux fichiers textes, car, à-dire-vrai, vous les avez déjà vues. Voici quelques prototypes :

Code : Ada

```

procedure New_Line (File : in File_Type; Spacing : in Positive_Count := 1);
procedure New_Page (File : in File_Type);
procedure Put (File : in File_Type; Item : in Character);
procedure Put (File : in File_Type; Item : in String);
procedure Put_Line (File : in File_Type; Item : in String);

```

Vous devriez donc et déjà vous douter de ce que font ces instructions. New_Line() insère un saut de ligne dans notre fichier à l'emplacement du curseur (par défaut, mais le paramètre spacing permet d'insérer 2, 3, 4... sauts de lignes). New_Page() insère un saut de page. Les procédures Put() permettent d'insérer un caractère ou toute une chaîne de caractères. La procédure put_Line() insère une chaîne de caractères et un saut de ligne.

Autres opérations

Il existe également quelques fonctions ou procédures utilisables quels que soient les modes ! En voici quelques-unes pour se renseigner sur notre fichier :

- Pour connaître le mode dans lequel un fichier est ouvert (In_File, Out_File, Append_File). Permet d'éviter des erreurs de manipulation.

Code : Ada

```
function Mode (File : in File_Type) return File_Mode;
```

- Pour connaître le nom du fichier ouvert.

Code : Ada

```
function Name (File : in File_Type) return String;
```

- Pour savoir si un fichier est ouvert ou non. Pour éviter de fermer un fichier déjà fermé, ou d'ouvrir un fichier déjà ouvert, etc.

Code : Ada

```
function Is_Open (File : in File_Type) return Boolean;
```

Et en voici quelques autres pour déplacer le curseur ou se renseigner sur sa position. Pour information, le type Positive_Count est en fait des natural mais 0 est exclu (les prototypes ci-dessous sont écrits tels qu'ils le sont dans le package Ada.Text_IO).

- Pour placer le curseur à une certaine colonne ou à une certaine ligne.

Code : Ada

```

procedure Set_Col (File : in File_Type; To : in Positive_Count);
procedure Set_Line (File : in File_Type; To : in Positive_Count);

```

- Pour connaître la colonne, la ligne ou la page où se situe le curseur.

Code : Ada

```

function Col (File : in File_Type) return Positive_Count;
function Line (File : in File_Type) return Positive_Count;
function Page (File : in File_Type) return Positive_Count;

```

Les fichiers binaires séquentiels



On ne peut manipuler que des fichiers contenant du texte ? Comment je fais si je veux enregistrer les valeurs contenues dans un tableau par exemple ? Je dois absolument utiliser les attributs `image` et `value` ?

Non, le fichier texte n'est pas obligatoire, mais il a l'avantage d'être modifiable par l'utilisateur grâce à un simple éditeur de texte, type Notepad par exemple. Mais si vous ne souhaitez qu'enregistrer des integer (par exemple), il est possible d'enregistrer vos données dans des **fichiers binaires** prévus à cet effet.



C'est quoi un fichier binaire ?

C'est un fichier constitué d'une suite de bits (les chiffres 0 et 1, seuls chiffres connus de l'ordinateur). Ça ne nous aide pas beaucoup ? Ça tombe bien car ces fichiers contiennent une suite d'informations illisibles par l'utilisateur. Pour pouvoir lire ces données, il faut faire appel à un logiciel qui connaîtra le type d'informations contenues dans le fichier et qui saura comment les traiter. Par exemple :

- Les fichiers MP3, sont illisibles par le commun des mortels. En revanche, en utilisant un logiciel tel VLC ou amaroK, vous pourrez écouter votre morceau de musique.
- Les fichiers images (bmp, tiff, png, jpg, gif...) ne seront lisibles qu'à l'aide d'un logiciel comme photofiltre par exemple.
- On pourrait encore multiplier les exemples, la plupart des fichiers utilisés aujourd'hui étant des fichiers binaires.

Parmi ces fichiers binaires, il en existe deux sortes : les **fichiers séquentiels** et les **fichiers à accès direct**. La différence fondamentale réside dans la façon d'accéder aux données qu'ils contiennent. Les premiers sont utilisables comme les fichiers textes (qui sont eux-mêmes des fichiers séquentiels), c'est pourquoi nous allons les aborder de ce pas. Les seconds sont utilisés un peu différemment, et nous les verrons dans la sous-partie suivante.

Nous allons créer un programme Enregister qui, comme son nom l'indique, enregistre des données de type integer dans un fichier binaire à accès séquentiel. Nous devons d'abord régler le problème des packages. Celui que nous allons utiliser ici est Ada.Sequential_IO. Le souci, c'est qu'il est fait pour tous les types de données (float, natural, array, string, integer, character...), on dit qu'il est générique (nous aborderons cette notion dans la quatrième partie de ce cours). La première chose à faire est donc de créer un package plus spécifique :

Code : Ada

```
WITH Ada.Integer_Text_IO ;           USE Ada.Integer_Text_IO ;
WITH Ada.Sequential_IO ;
```

```
PROCEDURE Enregistrer IS
  PACKAGE P_integer_file IS NEW Ada.Sequential_IO(integer) ;
  USE P_integer_file ;
  ...
```

Comme vous pouvez le remarquer, Ada.Sequential_IO ne bénéficie pas d'une clause `USE` (du fait de sa généréricité) et doit être « redéclarée » plus loin : nous créons un package `P_integer_file` qui est en fait le package Ada.Sequential_IO mais réservé aux données de type integer. Nous pouvons ensuite écrire notre clause de contexte « `USE P_integer_file` ». Nous avons déjà vu ce genre de chose durant notre premier TP pour créer des nombres aléatoires. Je ne vais pas m'embêter davantage sur cette notion un peu compliquée car nous y reviendrons plus tard. Notez tout de même la nomenclature : je commence mon nom de package par un `P_` pour mieux le distinguer des types ou des variables.

Nous allons ensuite déclarer notre variable composite fichier, notre type `T_Tableau` et notre variable `T` de type `T_Tableau`. Je vous laisse le choix dans la manière de saisir les valeurs de `T`. Puis, nous allons devoir créer notre fichier, l'ouvrir et le fermer :

Code : Ada

```
...
  MonFichier : File_Type ;
  type T_Tableau is array(1..5,1..5) of integer ;
  T : T_Tableau ;
BEGIN
  -- Saisie libre de T (débrouillez-vous) --
  -- enregistrement (vu après) --
  ...
  create(MonFichier,Out_File,"sauvegarde.inf") ;
  ...
  close(MonFichier) ;
END Enregistrer ;
```

Comme je vous l'ai dit, l'encart « *Saisie libre de T* » est à votre charge (affectation au clavier, aléatoire ou pré-déterminée, à vous de voir). La procédure `create()` peut être remplacée par la procédure `open()`, comme nous en avions l'habitude avec les fichiers textes. Les modes d'ouverture (`In_File`, `Out_File`, `Append_File`) sont également les mêmes. Concentrons-nous plutôt sur le deuxième encart : comment enregistrer nos données ? Les procédures `put_line()` et `put()` ne sont plus utilisables ici ! En revanche, elles ont une « sœur jumelle » : `write()` ! Traduction pour les anglophones : « *écrire* ». Notre second encart va donc pouvoir être remplacé par ceci :

Code : Ada

```
...
  for i in T'range(1) loop
    for j in T'range(2) loop
      write(MonFichier,T(i,j)) ;
    end loop ;
  end loop ;
  ...
```

Compiler votre code et exécutez-le. Un fichier « `sauvegarde.inf` » doit avoir été créé. Ouvrez-le avec le bloc-notes : il affiche des symboles bizarres, mais rien de ce que vous avez enregistré ! Je vous rappelle qu'il s'agit d'un fichier binaire ! Donc il est logique qu'il ne soit pas lisible avec un simple éditeur de texte. Pour le lire, nous allons créer, dans le même répertoire, un second programme : `Lire`. Il faudra donc « redéclarer » notre package `P_integer_file` et tout et tout. 😊

Pour la lecture, nous ne pouvons, bien entendu, pas non plus utiliser `get()` ou `get_line()`. Nous utiliserons la procédure `read()` (*traduction : lire*) pour obtenir les informations voulues et la fonction `Enf.Of.File()` pour être certains de ne pas saisir de valeur une fois rendu à la fin du fichier (à noter que les fins de ligne ou fins de page n'existe pas dans un fichier binaire).

Code : Ada

```
WITH Ada.Integer_Text_IO ;           USE Ada.Integer_Text_IO ;
WITH Ada.Sequential_IO ;
```

```
PROCEDURE Lire IS
  PACKAGE P_integer_file IS NEW Ada.Sequential_IO(integer) ;
  USE P_integer_file ;
  MonFichier : File_Type ;
  type T_Tableau is array(1..5,1..5) of integer ;
  T : T_Tableau ;
BEGIN
  open(MonFichier,In_File,"sauvegarde.inf") ;
  for i in T'range(1) loop
    for j in T'range(2) loop
      exit when Enf.Of.File(MonFichier) ;
      read(MonFichier,T(i,j)) ;
      put(T(i,j)) ;
    end loop ;
  end loop ;
  close(MonFichier) ;
END Lire ;
```



Les procédures et fonctions `delete()`, `reset()`, `Name()`, `Mode()` ou `Is_Open()` existent également pour les fichiers séquentiels.

Voilà, nous avons donc et déjà fait le tour des fichiers binaires à accès séquentiel. Le principe n'est guère différent des fichiers textes (qui sont, je le rappelle, eux aussi à accès séquentiel), retenez simplement que `put()` est remplacé par `write()`, que `get()` est remplacé par `read()` et qu'il faut préalablement spécifier le type de données enregistrées dans vos fichiers. Nous allons maintenant attaquer les fichiers binaires à accès direct. 😊

Les fichiers binaires directs

Les fichiers binaires à accès direct fonctionnent à la manière d'un tableau. Chaque élément enregistré dans le fichier est doté d'un numéro, d'un indice, comme dans un tableau. Tout accès à une valeur que ce soit en lecture ou en écriture, se fait en indiquant cet indice. Première conséquence et première grande différence avec les fichiers séquentiels, le mode `Append_File` n'est plus disponible. En revanche, un mode `InOut_File` est enfin disponible !

Nous allons enregistrer cette fois les valeurs contenues dans un tableau de float unidimensionnel. Nous utiliserons donc le package `Ada.Direct_IO` qui, comme `Ada.Sequential_IO`, est un package générique (fait pour tous types de données). Il nous faudra donc créer un package comme précédemment avec `P_integer_file` (mais à l'aide de `Ada.Direct_IO` cette fois). Voici le code de ce programme :

Code : Ada

```
WITH Ada.Float_Text_IO ;           USE Ada.Float_Text_IO ;
WITH Ada.Direct_IO ;
```

```
PROCEDURE FichierDirect IS
  PACKAGE P_float_file IS NEW Ada.Direct_IO(Float) ;
  USE P_float_file ;
  MonFichier : File_Type ;
  TYPE T_Vecteur IS ARRAY(1..8) OF Float ;
  X : T_Vecteur := (1.0,2.0,4.2,6.5,10.0,65.2,5.3,101.01) ;
  COUNT : float ;
BEGIN
  create(MonFichier,InOut_File,"vecteur.inf") ;
  for i in 1..8 loop
    write(MonFichier,T(i),count(i)) ;
    read(MonFichier,x,count(i)) ;
    put(x) ;
  end loop ;
  close(MonFichier) ;
END FichierDirect ;
```



Le troisième paramètre des procédures `write()` et `read()` est bien entendu l'indice de la valeur lue ou écrite. Toutefois, cet indice est de type `Count` (une sorte de type `natural`) incompatible malheureusement avec le type de la variable `i` (`integer`). C'est pourquoi nous sommes obligés d'écrire `count(i)` pour la convertir en variable de type `Count`.

Là encore, si vous tentez d'ouvrir votre fichier « `vecteur.inf` », vous aurez droit à un joli texte incompréhensible. Remarquez que l'instruction « `write(MonFichier,T(i),count(i))` » pourrait être remplacée par les deux lignes suivantes :

Code : Ada

```
set_index(MonFichier,count(i));
write(MonFichier,T(i));
```

En effet, l'instruction `set_index` place le curseur à l'emplacement voulu, et il existe une deuxième procédure `write()` qui ne spécifie pas l'indice où vous souhaitez écrire. Cette procédure écrit donc là où se trouve le curseur. Et il en est de même pour la procédure `Read()`.



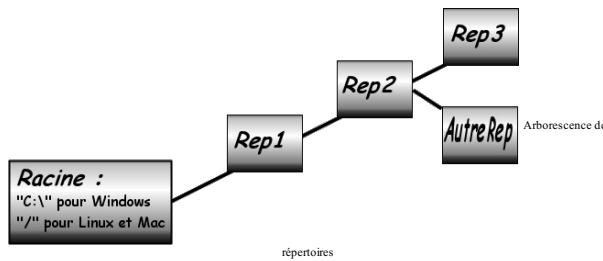
Comme pour un tableau, vous ne pouvez pas lire un emplacement vide, cela générera une erreur ! C'est là l'inconvénient des fichiers binaires à accès directs.

Les répertoires

Peut-être avez-vous remarqué que, comme pour les packages, les fichiers que vous créez ou que vous pouvez lire se situent tous dans le même répertoire que votre programme. Si jamais vous les déplacez, votre programme ne les retrouvera plus. Pourtant, lorsque vous mènerez des projets plus conséquents (comme les TP de ce tutoriel ou bien vos propres projets), il sera préférable de rassembler les fichiers de même types dans d'autres répertoires.

Chemins absolus et relatifs

Vous savez que les répertoires et fichiers de votre ordinateur sont organisés en arborescence. Chaque répertoire est ainsi doté d'une adresse, c'est à dire d'un chemin d'accès. Si votre programme est situé dans le répertoire `C:\Rep1\Rep2\Rep3` (utilise un chemin windows mais le principe est le même sous les autres systèmes d'exploitation), vous pouvez fournir deux adresses à l'ordinateur : soit cette adresse complète, appelée **chemin absolu**, soit le chemin à suivre pour retrouver les fichiers à partir de son répertoire courant, appelé **chemin relatif**.



Si votre programme est situé dans le même répertoire que vos fichiers, le chemin relatif sera noté par un point (.), que vous soyez sous Windows, Mac ou Linux. Ce point signifie « *dans le répertoire courant* ». Si vos fichiers sont situés dans le répertoire `C:\Rep1\Rep2` (adresse absolue), le chemin relatif sera noté par deux points (..) qui signifient « *dans le répertoire du dessous* ».

Compliquons la tâche : notre programme est toujours situé dans le répertoire `C:\Rep1\Rep2\Rep3` mais les fichiers sont situés dans le répertoire `C:\Rep1\Rep2\Rep3\Rep4`. L'adresse relative des fichiers est donc ..\Rep4. Enfin, si nous déplaçons nos fichiers dans un répertoire `C:\Rep1\Rep2\AutreRep`, leur chemin relatif sera ..\AutreRep (le programme doit revenir au répertoire du dessous avant d'ouvrir le répertoire AutreRep)

Indiquer le chemin d'accès

Maintenant que ces rappels ont été faits, créons donc un répertoire « `files` » dans le même répertoire que notre programme et plâsons-y un fichier appelé « `Sauvegarde.txt` ». Comment le lire désormais ? Au lieu que votre programme n'utilise l'instruction suivante :

Code : Ada

```
Open(F, In_File, "Sauvegarde.txt");
```

Nous allons lui indiquer le nom du fichier à ouvrir ainsi que le chemin à suivre :

Code : Ada

```
Open(F, In_File, "./files/Sauvegarde.txt"); --Avec adresse relative
-- OU
Open(F, In_File, "C:/Rep1/Rep2/Rep3/files/Sauvegarde.txt"); --Avec
adresse absolue
```

Le paramètre pour le nom du fichier contient ainsi non seulement son nom mais également son adresse complète.

Gérer fichiers et répertoires



Le programme ne pourra pas copier ou supprimer les répertoires lui-même ?

Bien sûr que si ! Le langage Ada a tout prévu. Pour cela, vous aurez besoin du package `Ada.Directories`. Le nom du fichier correspondant est `a-direct.adb` et je vous invite à le lire par vous-même pour découvrir les fonctionnalités que je ne détaillerai pas ici. Voici quelques instructions pour gérer vos répertoires :

Code : Ada

```
function Current_Directory return String;
procedure Set_Directory (Directory : String);
procedure Create_Directory (New_Directory : String);
procedure Delete_Directory (Directory : String);
procedure Create_Path (New_Directory : String);
procedure Delete_Tree (Directory : String);
```

`Current_Directory` renvoie une chaîne de caractères correspondant à l'adresse du répertoire courant. La procédure `Set_Directory` permet quant à elle de définir un répertoire par défaut, si par exemple l'ensemble de vos ressources se trouvait dans un même dossier. `Create_Directory()` et `Delete_Directory()` vous permettront de créer ou supprimer un répertoire unique, tandis que `Create_Path()` et `Delete_Tree()` vous permettront de créer toute une chaîne de répertoire ou de supprimer un dossier et l'ensemble de ses sous-dossiers. Des programmes similaires sont disponibles pour les fichiers :

Code : Ada

```
procedure Delete_File (Name : String);
procedure Rename (Old_Name, New_Name : String);
procedure Copy_File (Source_Name : String;
                     Target_Name : String);
```

Ces procédures vous permettront respectivement de supprimer un fichier, de le renommer ou de le copier. Pour les procédures `Rename()` et `Copy_File()`, le premier paramètre correspond au nom actuel du fichier, le second paramètre correspond quant à lui au nouveau nom ou au nom du fichier cible. Voici enfin trois dernières fonctions parmi les plus utiles :

Code : Ada

```
function Size (Name : String) return File_Size;
function Exists (Name : String) return Boolean;
function Kind (Name : String) return File_Kind;
```

`Size()` renverra la taille d'un fichier ; `Exists()` vous permettra de savoir si un fichier ou un dossier existe ; `Kind()` enfin, renverra le type de fichier visé : un dossier (le résultat vaudra alors « `Directory` »), un fichier ordinaire (le résultat vaudra alors « `Ordinary_File` ») ou un fichier spécial (le résultat vaudra alors « `Special_File` »).

Quelques exercices

Exercice 1

Énoncé

Créer un fichier texte appelé "`poeme.txt`" et dans lequel vous copierez le texte suivant :

[Secret \(cliquez pour afficher\)](#)

```
Que j'aime à faire apprendre un nombre utile aux sages !
Immortel Archimède, artiste, ingénieur,
Qui de ton jugement peut priser la valeur ?
Pour moi ton problème eut de pareils avantages.

Jadis, mystérieux, un problème bloquaît
Tout l'admirable procédé, l'œuvre grandios.
Que Pythagore découvrit aux anciens Grecs.
Ô quadrature ! Vieux tourment du philosophe

Insoluble rondeur, trop longtemps vous avez
Defie Pythagore et ses imitateurs.
Comment intégrer l'espace plan circulaire ?
Former un triangle auquel il équivaudra ?

Nouvelle invention : Archimède inscrira
Dedans un hexagone ; appréciera son aire
Fonction du rayon. Pas trop ne s'y tiendra :
Dédoublez chaque élément antérieur ;

Toujours de l'orbe calculée approchera ;
Définira limite ; enfin, l'arc, le limiteur
De cet inquiétant cercle, ennemi trop rebelle
Professeur, enseignez son problème avec zèle
```

Ce poème est un moyen mnémotechnique permettant de réciter les chiffres du nombre π ($\approx 3,14159$), il suffit pour cela de compter le nombre de lettres de chacun des mots (10 équivalant au chiffre 0).

Puis, créer un programme `poeme.exe` qui lira ce fichier et l'affichera dans la console.

[Solution](#)

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
with ada.Text_IO, ada.Strings.Unbounded ;
use ada.Text_IO, ada.Strings.Unbounded ;

procedure poeme is
  F : File_Type ;
  txt : unbounded_string ;
begin
  open(F,in_file, "poeme.txt") ;
  while not end_of_file(F) loop
    txt :=to_unbounded_string(get_line(F)) ;
    put_line(to_string(txt)) ;
  end loop ;
  close(F) ;
end poeme ;
```

Exercice 2

Énoncé

Plus compliqué, reprendre le problème précédent et modifier le code source de sorte que le programme affiche également les chiffres du nombre π . Attention ! La ponctuation ne doit pas être comptée et je rappelle que la seule façon d'obtenir 0 est d'avoir un nombre à 10 chiffres !

[Solution](#)

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
WITH Ada.Text_IO, Ada.Integer_Text_IO, Ada.Strings.Unbounded ;
USE Ada.Text_IO, Ada.Integer_Text_IO, Ada.Strings.Unbounded ;

PROCEDURE Poeme IS
  F : File_Type;
  Txt : Unbounded_String;

  PROCEDURE Comptage (Txt : String) IS
    N : Natural := 0;
  BEGIN
    FOR I IN Txt'RANGE LOOP
      CASE Txt(I) IS
        WHEN '0' .. '9' => IF N/=0 THEN Put(;
        WHEN ' ' | '?' | ',' | ';' | '!' | ':' | '''' => IF N/=0 THEN Put(;
        WHEN others => n := n+1 ;
      END CASE ;
    END LOOP ;
  END Comptage ;

  BEGIN
    Open(F,In_File, "poeme.txt") ;
    WHILE NOT End_Of_File(F) LOOP
      Txt :=To_Unbounded_String(Get_Line(F)) ;
      Put_Line(To_String(Txt)) ;
      Comptage(To_String(Txt)) ;
      New_Line ;
    END LOOP ;
    Close(F) ;
  END Poeme ;
```

Exercice 3

Énoncé

Nous allons créer un fichier "`Highest_Score.txt`" dans lequel seront enregistrés les informations concernant le meilleur score obtenu à un jeu (ce code pourra être mis en package pour être réutilisé plus tard). Seront demandés : le nom du gagnant, le score obtenu, le jour, le mois et l'année d'obtention de ce résultat.

Voici un exemple de la présentation de ce fichier texte. Pas d'espaces, les titulés sont en majuscule suivis d'un signe =. Il y aura 4 valeurs entières à enregistrer et un string.



Le programme donnera le choix entre lire le meilleur score et en enregistrer un nouveau. Vous aurez besoin pour cela du package Ada.Calendar et de quelques fonctions :

- clock : renvoie la date du moment. Résultat de type Time.
- year(clock) : extrait l'année de la date du moment. Résultat : integer.
- month(clock) : extrait le mois de la date du moment. Résultat : integer.
- day(clock) : extrait le jour de la date du moment. Résultat : integer.

Solution
Secret (cliquez pour afficher)

```
Code : Ada

WITH Ada.Text_IO ;           USE Ada.Text_IO ;
WITH Ada.Integer_Text_IO ;   USE Ada.Integer_Text_IO ;
WITH Ada.Calendar;          USE Ada.Calendar ;
WITH Ada.Characters.Handling ; USE Ada.Characters.Handling ;
WITH Ada.Strings.Unbounded ; USE Ada.Strings.Unbounded ;

procedure BestScore is

    -- Lecture du score --
    procedure read_score(F : file_type) is
        txt : unbounded_string ;
        long : natural ;
        Score, Jour, Mois, Annee : natural ;
        Nom : unbounded_string ;
    begin
        while not end_of_file(F) loop
            txt:=To_Unbounded_String(get_line(F)) ;
            long := length(txt) ;
            if long >= 6 then
                if To_String(txt)(1..6)="SCORE=" then
                    Score := natural'value(To_String(txt)(7..long)) ;
                end if ;
                if To_String(txt)(1..6)="MONTH=" then
                    Mois := natural'value(To_String(txt)(7..long)) ;
                end if ;
            end if ;
            if long >= 5 then
                if To_String(txt)(1..5)="YEAR=" then
                    Annee := natural'value(To_String(txt)(6..long)) ;
                end if ;
                if To_String(txt)(1..5)="NAME=" then
                    Nom := to_unbounded_string(To_string(txt)(6..long)) ;
                end if ;
            end if ;
            if long >= 4 and then To_String(txt)(1..4)="DAY=" then
                Jour := natural'value(To_String(txt)(5..long)) ;
            end if ;
        end loop ;
        Put_line(" " & to_string(Nom)) ;
        Put("a obtenu le score de ") ; Put(Score) ; New line ;
        Put("le " & integer'image(jour) & "/" & integer'image(mois) & "/" & integer'image(annee)) ;
    end read_score ;

    -- Ecriture du score --
    procedure Write_Score(F : file_type) is
        txt : unbounded_string ;
        score : integer ;
    begin
        Put("Quel est votre nom ? ") ; txt := to_unbounded_string(get_line) ;
        Put_line(F,"NAME=" & to_string(txt)) ;
        Put("Quel est votre score ? ") ; get(score) ; skip_line ;
        Put_line(F,"SCORE=" & integer'image(score)) ;
        Put_line(F,"YEAR=" & integer'image(year(clock))) ;
        Put_line(F,"MONTH=" & integer'image(month(clock))) ;
        Put_line(F,"DAY=" & integer'image(day(clock))) ;
    end Write_Score ;

    -- PROCÉDURE PRINCIPALE --
    c : character ;
    NomFichier : constant string := "Highest_Score.txt" ;
    F : file_type ;
begin
begin
    loop
        Put_line("Que voulez-vous faire ? Lire (L) le meilleur score obtenu, ");
        Put("ou en enregistrer un nouveau (N) ? ") ;
        get_immediate(c) ; new_line ; new_line ;
        c := to_upper(c) ;
        exit when c = 'L' or c = 'N' ;
    end loop ;
    if c = 'L'
        then open(F,In_File,NomFichier) ;
            read_score(F) ;
            close(F) ;
        else create(F,Out_File,NomFichier) ;
            write_score(F) ;
            close(F) ;
        open(F,In_File,NomFichier) ;
            read_score(F) ;
            close(F) ;
    end if ;
end BestScore ;
```

Ce chapitre sur les fichiers vous permettra enfin de garder une trace des actions de vos programmes ; vous voilà désormais armés pour créer des programmes plus complets. Vous pouvez dès lors déjà réinvestir ce que vous avez vu et conçu dans ces derniers chapitres pour améliorer votre jeu de craps (vous vous souvenez, le TP) en permettant au joueur de sauvegarder son (ses) score(s), par exemple. Pour le prochain chapitre, nous allons mettre de côté les tableaux (vous les retrouverez vite, rassurez-vous !) pour créer nos propres types.

En résumé :

- Avant d'effectuer la moindre opération sur un fichier, ouvrez-le et surtout, n'oubliez pas de le fermer avant de clore votre programme.
- Les fichiers séquentiels (que ce soient des fichiers textes ou binaires) ne sont accessibles qu'en lecture ou en écriture, mais jamais les deux en même temps.
- Il est préférable de ne pas garder un fichier ouvert trop longtemps, on préfère en général copier son contenu dans une ou plusieurs variables et travailler sur ces variables.
- Avant de saisir une valeur en provenance d'un fichier, assurez-vous que vous ne risquez rien : vous pouvez être en fin de ligne, en fin de page ou en fin de fichier ; dans un fichier à accès direct, l'emplacement suivant n'est pas nécessairement occupé !
- N'hésitez pas à classer vos fichiers dans un sous-dossier par soucis d'organisation. Mais pensez alors à indiquer le chemin d'accès à votre programme, de préférence un chemin relatif.

Créez vos propres types

Nous avons déjà abordé de nombreux types (Integer, Float, Character, String, tableaux et j'en passe) mais peu d'entre eux suffisent à modéliser la plupart des objets de la vie courante. Imaginez que vous voulez modéliser l'identité des personnages de votre futur jeu vidéo révolutionnaire : des String modéliseraient les nom et prénom, un Integer peut modéliser l'âge, des Float pourraient modéliser la taille et le poids... mais aucun type ne peut à lui seul modéliser l'ensemble de ces données, même pas les tableaux. De même, comment modéliser une pendule ? Pour les Integer, $23 + 2 = 25$, alors que pour une pendule $23H + 2H$ donne $1H$ du matin !

Nous allons donc apprendre à créer nos propres types composés, à les modifier, à les faire évoluer et même à créer des «sous-types» !

Créez à partir de types préédéfinis

Cette partie n'est pas la plus folichonne : nous allons réutiliser des types préexistants pour créer des nouveaux. Malgré tout, elle vous sera souvent très utile et elle n'est pas très compliquée à comprendre.

Sous-type comme intervalle

Schématiquement, Integer couvre les nombres entiers ($0, 1, 2, 3, \dots, -1, -2, -3, \dots$) et natural les nombres entiers naturels ($0, 1, 2, 3, \dots$). Nous souhaiterions créer un type *T_positif* et un type *T_négatif* qui ne comportent pas le nombre 0. Pour cela, nous allons créer un sous-type (**SUBTYPE**) de natural ou de integer qui couvrira l'intervalle (**RANGE**) allant de 1 à l'infini (symbole : $+\infty$) pour les positifs et de «l'infini négatif» (symbole : $-\infty$) à -1 pour les négatifs.

 L'infini n'existe pas en informatique. Les nombres générés sont nécessairement limités, quand bien même cette limite est très élevée, il ne s'agit pas de l'infini au sens mathématique.

Nous allons donc, dans la partie déclarative, écrire soit :

Code : Ada

```
subtype T_positif is integer range 1..integer'last;
subtype T_négatif is integer range integer'first..-1;
```

Soit :

Code : Ada

```
subtype T_positif is natural range 1..natural'last;
```

Il sera alors impossible d'écrire ceci :

Code : Ada

```
n : T_positif := 0;
m : T_négatif := 1;
```

Comme *T_positif* et *T_négatif* dérivent des types *integer* ou *natural* (lui-même dérivant du type *integer*), toutes les opérations valables avec les *integer/natural*, comme l'addition ou la multiplication, sont bien sûr valables avec les types *T_positif* et *T_négatif*, tant que vous ne sortez pas des intervalles définis.

Types modulaires

Deuxième exemple, nous souhaiterions créer un type *T_chiffre* allant de 0 à 9. Nous écrirons alors :

Code : Ada

```
subtype T_chiffre is integer range 0..9;
```

Seulement, les opérations suivantes engendreront une erreur :

Code : Ada

```
...
c : T_chiffre;
BEGIN
  c:= 9;      --Jusque là, tout va bien
  c:= c+2;    --c devrait valoir 11. Aie ! Pas possible !
  ...

```

Il serait donc plus pratique si après 9, les comptes recommençaient à 0 ! Dans ce cas $9+1$ donnerait 0, $9+2$ donnerait 1, $9+3$ donnerait 2... C'est à cela que servent les **sous-types modulaires** que l'on définit ainsi :

Code : Ada

```
type T_chiffre is mod 10;
```

Le type *T_chiffre* comportera 10 nombres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ! Pour ceux qui ont de bonnes connaissances en mathématiques, $X=9$ équivaudra à écrire $X \equiv 9(10)$ c'est-à-dire que « X est congru à 9 modulo 10 ».

Autre exemple, nous allons créer des types *T_Heure*, *T_Minute*, *T_Seconde*, *T_Milliseconde* pour indiquer le temps. Remarquez que depuis le début, j'introduis mes types par la lettre *T*, ce qui permet de distinguer rapidement si un mot correspond à un type ou à une variable.

Code : Ada

```
type T_Heure is mod 24;
type T_Minute is mod 60;
type T_Seconde is mod 60;
type T_Milliseconde is mod 1_000          --
  on peut aussi écrire 1000, mais le symbole _ (touche 8)
  --
  permet de séparer les chiffres pour y voir plus clair
```



Le nombre écrit après MOD n'est pas le nombre maximal du sous-type mais le nombre d'éléments contenus dans ce sous-types, le premier étant 0. Par exemple, ces déclarations interdisent donc à une variable *H* de type *T_Heure* de valoir 24 ! Si elle vaut 23 et qu'on l'incrémenté (+1) alors elle retournera à 0. De même une variable *mil* de type *T_Milliseconde* peut prendre une valeur entre 0 et 999 mais pas 1000 !

Une variable *M* de type *T_Minute* n'est plus un *integer* ou un *natural* ! C'est un type modulaire au fonctionnement un peu différent. Pour l'afficher, par exemple, il faudra écrire :



```
Put(integer(M));
```

Énumérez les valeurs d'un type

Continuons avec notre exemple du temps. Les types *T_Heure*, *T_Minute*, *T_Seconde* et *T_Milliseconde* ont été créés comme des types modulaires. Créer un type *T_Annee* ne devrait pas vous poser de soucis. Très bien. Mais pour les jours ou les mois, il serait bon de distinguer trois types distincts :

- *T_NumerO_Jour* : type allant de 1 à 31 (non modulaire car commençant à 1 et non 0)
- *T_Nom_Jour* : type égal à LUNDI, MARDI, MERCREDI...
- *T_Mois* : type égal à JANVIER, FEVRIER, MARS...

Nous savons déclarer le premier type :

Code : Ada

```
subtype T_NumerO_Jour is integer range 1..31;
```

Pour les suivants, il faut créer ce que l'on appelle un type énuméré (un type où il faut énumérer toutes les valeurs disponibles), de la manière suivante :

Code : Ada

```
type T_Mois is (JANVIER,FEVRIER,MARS,AVRIL,MAI,JUIN,
                 JUILLET,AOUT,SEPTEMBRE,OCTOBRE,NOVEMBRE,DECEMBRE) ;
type T_Nom_Jour is (LUNDI,MARDI,MERCREDI,JEUDI,VENDREDI,SAMEDI,DIMANCHE) ;
```



Pourquoi ne pas se contenter d'un nom pour représenter le jour ou le mois ?

Tout simplement par souci de clarté et de mémorisation. Est-il plus simple d'écrire : «jour = 4» ou «jour = JEUDI» ? «mois = 5» ou «mois := MAI» ? La deuxième façon est bien plus lisible. Souvenez-vous des fichiers et de leur mode d'ouverture : ne vaut-il pas mieux écrire «In_File», «Out_File» ou «Append_File» que 0, 1 ou 2 ? Ne vaut-il mieux pas écrire **TRUE** ou **FALSE** que 1 ou 0 ? De plus, il est possible d'utiliser les attributs avec les types énumérés :

- `T_Mois'first` renverra JANVIER (le premier mois)
- `T_Nom_Jour'last` renverra DIMANCHE (le dernier jour)
- `T_Mois'succ(JUIN)` renverra JUILLET (le mois suivant JUIN)
- `T_Mois'pred(JUIN)` renverra MAI (le mois précédent JUIN)
- `T_Mois'pos(JUIN)` renverra 5 (le numéro du mois de JUIN dans la liste)
- `T_Mois'val(1)` renverra FEVRIER (le mois n°1 dans la liste, voir après)
- `put(T_Nom_Jour'image(MARDI))` permettra d'écrire MARDI à l'écran



Les types énumérés sont rangés à partir de 0, donc JANVIER est le mois n°0 et FÉVRIER le mois n°1 !

Par conséquent, il est même possible de comparer des variables de type énuméré : MARDI>JEUDI renverra false ! Enfin, les affectations se font de la manière la plus simple qu'il soit :

Code : Ada

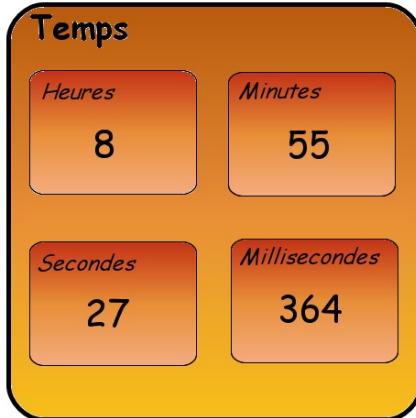
```
...
mois : T_Mois ;
BEGIN
mois := AOUT ;
```

Les types structurés

Déclarer un type «construit»

La théorie

Nous disposons maintenant des types `T_Heure`, `T_Minute`, `T_Seconde`, `T_Milliseconde`, `T_Annee` (je vous ai laissé libres de le faire), `T_Mois`, `T_Nom_Jour` et `T_Numero_Jour`, ouf ! C'est un peu long tout ça, non ? ☺ Il serait plus pratique de tout mettre dans un seul type de données (`T_Temps` ou `T_Date` par exemple) de manière à éviter de déclarer tout un escadron de variables pour enregistrer une simple date.



Un type `T_Temps` permettrait de

rassembler toutes les informations

C'est ce que nous permettent les **types articles**. Si le terme employé par la terminologie Ada est « Type article », bon nombre de langages parlent plutôt de **types structurés** ou (d'ailleurs en C, le mot clé est **struct**). Pour une fois les termes du C me semblent plus clairs et je préférerais donc généralement cette terminologie. ☺ Nous allons donc créer notre type structuré en écrivant « **TYPE T_Temps IS** » comme d'habitude puis nous allons ouvrir un bloc **RECORD**. L'exemple par l'image :

Code : Ada

```
type T_Temps is
record
  ...
end record ;
```

Rappelons que «*to record*» signifie en anglais «Enregistrer». C'est donc entre les instructions **RECORD** et **END RECORD** que nous allons enregistrer le contenu du type `T_Temps`, c'est-à-dire l'heure, les minutes, les secondes et les millisecondes :

Code : Ada

```
type T_Temps is
record
  H : T_Heure ;
  Min : T_Minute ;
  S : T_Seconde ;
  Milli : T_Milliseconde ;
end record ;
```

Ainsi, c'est comme si l'on déclarait quatre variables de types différents sauf que ce que l'on déclare, ce sont les quatre **composantes** de notre type. Par la suite, nous n'aurons plus qu'une seule variable de type `T_Temps` à déclarer et à modifier. Il est même possible d'attribuer des valeurs par défaut à nos différentes composantes. Ainsi, contrairement aux autres types de variables ou d'objets prédéfinis, il sera possible d'initialiser un objet de type `T_Temps` simplement en le déclarant.

Code : Ada

```
type T_Temps is
record
  H : T_Heure := 0 ;
  Min : T_Minute := 0 ;
  S : T_Seconde := 0 ;
  Milli : T_Milliseconde := 0 ;
end record ;
```

t : T_Temps ; --
Sans rien faire, t vaut automatiquement 0 H 0 Min 0 S 0 ms !

Mise en pratique

Voici quelques petits exercices d'application bêtes et méchant.

- Créer un type T_Jour contenant le nom et le numéro du jour.
- Créer le type T_Date contenant le jour, le mois et l'année.

Solution

Secret (cliquez pour afficher)

Code : Ada

```
type T_Jour is
  record
    nom : T_Nom_Jour := LUNDI;
    num : T_Numero_Jour := 1;
  end record;

type T_Date is
  record
    jour : T_Jour;
    mois : T_Mois;
    annee : T_Annee;
  end record;
```

Ordre des déclarations

Il est très important de déclarer vos types dans l'ordre suivant :

1. les types : T_Annee, T_Mois, T_Nom_Jour, T_Numero_Jour dans n'importe quel ordre ;
2. le type T_Jour ;
3. le type T_Date !

En effet, le type T_Jour a besoin pour être déclaré que les types T_Numero_Jour et T_Nom_Jour soient déjà déclarés, sinon, lors de la compilation, le compilateur va tenter de construire notre type T_Jour et découvrir que pour cela il a besoin de deux types inconnus ! Car le compilateur va lire votre code du haut vers le bas et il lève les erreurs au fur et à mesure qu'elles apparaissent.

Pour pallier à ce problème, soit vous respectez l'ordre logique des déclarations, soit vous ne voulez pas (c'est bête) ou vous ne pouvez pas (ça arrive) respecter cet ordre et auquel cas, je vous conseille d'écrire les spécifications des types nécessaires auparavant :

Code : Ada

```
type T_Nom_Jour;          --Liste des spécifications
                           évitant les soucis à la compilation
type T_Numero_Jour;
type T_Jour;

type T_Date is           --Définition des types
  (le tout en vrac)
  record
    ...
  end record;

type T_Jour is
  record
    ...
  end record;

type T_Nom_Jour is
  record
    ...
  end record;

type T_Numero_Jour is
  record
    ...
  end record;
```

Déclarer et modifier un objet de type structuré**Déclaration d'un objet de type structure**

Le langage Ada considère automatiquement les types que vous construisez comme les siens. Donc la déclaration se fait le plus simplement du monde, comme pour n'importe quelle variable :

Code : Ada

```
t : T_Temps;
prise_bastille : T_Date;
```

Affectation d'un objet de type structure

Comment attribuer une valeur à notre objet t de type T_Temps ? Il y a plusieurs façons. Tout d'abord avec les agrégats, comme pour les tableaux (à la différence que les valeurs ne sont pas toutes du même type) :

Code : Ada

```
t := (11,55,48,763); -- pensez à respecter l'ordre dans lequel les composantes
                       ont été déclarées lors du record : H,Min,S,Milli
t := (H => 11, Min => 55, Milli => 604, S => 37); -- Plus de soucis d'ordre
```

Cette méthode est rapide, efficace mais pas très lisible. Si on ne connaît pas bien le type T_Temps, on risque de se tromper dans l'ordre des composantes. Et puis, si on veut seulement faire une affectation sur les minutes ? Voici donc une seconde façon, sans agrégats :

Code : Ada

```
t.H := 11;
t.S := 48;
t.Min := 55;
t.Milli := 763;
...
t.Min := t.Min + 15; -- Possible car t.Min est la composante de type T_Minute qui est un type modulaire
En revanche, les t.H n'est pas incrémenté pour autant !
```

Un nouvel exercice et sa correction

À vous d'affecter une valeur à notre objet prise_bastille ! Pour rappel, la Bastille est tombée le 14 juillet 1789 (et c'était un mardi).

Secret (cliquez pour afficher)

Code : Ada

```
prise_bastille := ((MARDI,14),JUILLET,1789) ;
--OU BIEN
prise_bastille.jour.nom := MARDI ;
prise_bastille.jour.num := 14 ;
prise_bastille.mois := JUILLET ;
prise_bastille.annee := 1789 ;
```

Il faudra, pour la deuxième méthode, utiliser deux fois les points pour accéder à la composante de composante.

22 ! Rev'là les tableaux !

Compliquons notre tâche en mélangeant tableaux et dates !

Tableau dans un type

Tout d'abord, il pourrait être utile que certaines dates aient un nom ("Marignan", "La Bastille", "Gergovie"... enregistré dans un string de 15 cases (par exemple). Reprenons notre type T_Date :

Code : Ada

```
type T_Date is
record
    nom : string(1..15) := "Pas d'An 0 !!! ";
    jour : T_Jour := (LUNDI,1) ;
    mois : T_Mois := JANVIER;
    annee : T_Annee := 0 ;
end record ;
```

Il contient désormais un tableau (le string). Les opérations suivantes sont alors possibles :

Code : Ada

```
...
    prise_bastille : T_Date ;
BEGIN
    prise_bastille.nom := "La Mastique" ;      --Erreur volontaire,
    pas d'inquiétude ^^
    prise_bastille.nom(4) := 'E' ;             --Rectifications
    prise_bastille.nom(9..10) := "11" ;
```



Attention, prise_bastille n'est pas un tableau, c'est prise_bastille.nom qui en est un !

Type dans un tableau (Euh... Van Gogh ?)

Plus compliqué, on veut créer un tableau avec des dates à retenir (10 dates par exemple). Une fois notre type T_Date déclaré, déclarons notre type T_Chronologie et une variable T :

Code : Ada

```
type T_Chronologie is array(1..10) of T_Date ;
T : T_Chronologie ;
```

Cette fois c'est T qui est notre tableau et T(1), T(2)... sont des dates. Allons-y pour quelques affectations :

Code : Ada

```
T(1) := ("La Bastille", (MARDI,14),JUILLET,1789) ;
T(2).Nom := "Marignan" ;
T(2).Annee := 1515 ;
T(2).Jour.Num := 14 ;

T(3).Nom := "Jour G" ;
T(3).Nom(6) := 'J' ;
T(3).Nom(1..12) := "Debarquement" ;
```

Eh oui, les dernières sont bizarres, non ? Petite explication :

- T est un tableau ;
- T(3) est la 3ème case du tableau qui contient une date ;
- T(3).Nom est un String, soit un tableau contenant des caractères et contenu dans la troisième case du tableau T ;
- T(3).Nom(6) est la 6ème case du tableau T(3).Nom et c'est le caractère "G" que l'on modifie ensuite en "J".

Les types structurés : polymorphes et mutants !

Les types structurés polymorphes

Prérequis

La notion de type polymorphe était déjà présente dans la norme Ada83, la norme Ada95 a intégré la Programmation Orientée Objet pour laquelle le polymorphisme a un autre sens : le polymorphisme recouvre donc deux aspects en Ada. Nous parlerons pour l'instant du polymorphisme de type, par distinction avec le polymorphisme de classe cher à la POO.

Le **polymorphisme** est la propriété d'avoir plusieurs formes, plusieurs apparences. En informatique, c'est l'idée de permettre à un type donné, d'avoir plusieurs apparences distinctes. Nos types T_Date et T_Temps ne conviennent plus à cette partie, prenons un nouvel exemple : un type T_Bulletin qui contient les moyennes d'un élève dans différentes matières. Selon notre élève est en primaire, au collège ou au lycée, il n'aura pas les mêmes matières (nous n'entrerons pas dans le détail des filières et des options par classe, l'exemple doit rester compréhensible, quitte à être caricatural). Considérons les données suivantes :

- Un élève de primaire a du Français, des Maths et du Sport.
- Un élève de collège a du Français, des Maths, du Sport, une langue étrangère (LV1) et de la Biologie.
- Un élève de lycée a du Français, des Maths, du Sport, une Langue Etrangère (LV1), de la Biologie, de la Chimie et une deuxième Langue Etrangère (LV2).

Oui, je sais il manque beaucoup de choses et c'est très loin de la réalité, mais je n'ai pas envie d'écrire des dizaines de composantes !

Notre type T_Bulletin n'a pas besoin d'avoir une composante Chimie si notre élève est en primaire ; il doit donc s'adapter à l'élève ou plutôt à sa classe ! Nous allons donc définir deux autres types : un type énumératif T_Classe et un type structuré T_Elève.

Code : Ada

```
type T_Classe is (PRIMAIRE, COLLEGE, LYCEE) ;

type T_Elève is
record
    nom : string(1..20) ;           --20 caractères devraient suffire
    prénom : string(1..20) ;        --
    on commencera au besoin par des espaces
    classe : T_Classe ;           --
    Tout l'intérêt d'écrire un T devant le nom de notre type ! !
end record ;
```

Créer un type polymorphe T_Bulletin

Maintenant, nous allons pouvoir créer notre type `T_Bulletin`, mais comme nous l'avons dit, sa structure dépend de l'élève et surtout de sa classe. Il faut donc paramétrer notre type `T_Classe` !



Paramétriser un type ? C'est possible ?

Bien sûr, souvenez vous des tableaux ! Le type `ARRAY` est générique, il peut marcher avec des integer, des float... Il n'y a donc pas de raison que les types articles n'aient pas la même capacité. Pour notre type `T_Bulletin`, cela devrait donner ceci :

Code : Ada

```
type T_Bulletin(classe : T_Classe) is
record
    --PARTIE FIXE
    francais, math, sport : float; --ou float range 0.0..20.0;
    --PARTIE VARIABLE : COLLEGE ET LYCEE
    case classe is
        when PRIMAIRE =>
            null;
        when COLLEGE | LYCEE =>
            lvl, bio : float;
    end case;
end record;
```

Ce paramétrage nous permet ainsi d'éviter d'avoir à définir plusieurs types `T_Bulletin_Primary`, `T_Bulletin_College`, `T_Bulletin_Lycee`. Il suffit de fournir en paramètre la classe en question. Attention toutefois ! Le paramètre fourni doit être discret.



Parce qu'un paramètre peut-être bruyant ? 😊

Non, discret n'est pas ici le contraire de bruyant ! C'est une notion mathématique qui signifie que les valeurs sont toutes « isolées les unes des autres » ou, d'une autre façon, si je prends deux valeurs au hasard, il n'y a qu'un nombre fini de valeurs comprises entre les deux que j'ai prises. Par exemple : les integer sont de type discret (entre 2 et 5, il n'y a que 3 et 4, soit 2 nombres) ; les nombres réels (et le type float) ne sont pas discrets (entre 2.0 et 5.0, il y a une infinité de nombres comme 3.0 ou 3.01 ou 3.0000000002 ou 4.124578521...). De même, nos types énumérés sont discrets, mais les types structurés ne sont pas acceptés comme discrets !



Autre souci : c'était pas plus simple d'utiliser `IF / ELSIF` plutôt que des `CASE imbriqués` ?

Mais vous doutez bien que non ! Les conditions sont strictes lorsque l'on construit un type polymorphe. Pas de `IF`, ne pas faire de répétition dans l'instruction `WHEN` (ne pas répéter la composante `lv1` par exemple), un seul `CASE` et c'est fini (d'où l'obligation de les imbriquer plutôt que d'en écrire deux distincts)... C'est qu'on ne fera plus ce que l'on veut ! 🤪

Des objets polymorphes

Maintenant que nous avons déclaré nos types, il faut déclarer nos variables :

Code : Ada

```
Kevin : T_Elève := ("DUPONT", "Kevin", "PRIMAIRE");
Laura : T_Elève := ("DUPUIS", "Laura", "LYCEE");
Bltn_Kevin : T_Bulletin(Kevin.classe);
Bltn_Laura : T_Bulletin(Laura.classe);
```

Vous devriez avoir remarqué deux choses :

- Il faut fournir un paramètre quand on définit `Bltn_Kevin` et `Bltn_Laura`, sans quoi le compilateur échouera et vous demandera de préciser votre pensée.
- Par conséquent, lorsque vous déclarez les variables Kevin et Laura, vous devez les avoir initialisées (ou tout du moins avoir initialisé `Kevin.classe` et `Laura.classe`).

Nous avons ainsi obtenu deux variables composées `Bltn_Kevin` et `Bltn_Laura` qui n'ont pas la même structure. `Bltn_Kevin` a comme composantes :

- `Bltn_Kevin.classe` : accessible seulement en lecture, pour des tests par exemple.
- `Bltn_Kevin.Math`, `Bltn_Kevin.Français` et `Bltn_Kevin.sport` : accessibles en lecture et en écriture.

Quant à elle, la variable `Bltn_Laura` a davantage de composantes :

- `Bltn_Laura.classe` : en lecture seulement
- `Bltn_Laura.Math`, `Bltn_Laura.Français`, `Bltn_Laura.sport` : en lecture et écriture (comme `Bltn_Kevin`)
- `Bltn_Laura.lv1`, `Bltn_Laura.lv2`, `Bltn_Laura.bio`, `Bltn_Laura.chimie` : en lecture et écriture.



Le paramètre `Bltn_Laura.classe` se comporte comme une composante mais n'est pas modifiable ! Impossible d'écrire par la suite `Bltn_Laura.classe := PRIMAIRE!!!`

Il est ensuite possible de créer des sous-types (pour éviter des erreurs par exemple) :

Code : Ada

```
type T_Bulletin_Primary is T_Bulletin(PRIMAIRE);
type T_Bulletin_College is T_Bulletin(COLLEGE);
type T_Bulletin_Lycee is T_Bulletin(LYCEE);
```

Les types structurés mutants

Comment peut-on muter ?

Je vous arrête tout de suite : il ne s'agit pas de monstres ou de types avec des pouvoirs spéciaux ! Non, un type mutant est simplement un type structuré qui peut muter, c'est-à-dire changer. Je vais pour cela reprendre mon type `T_Bulletin`. Attention ça va aller très vite :

Code : Ada

```
type T_Bulletin(classe : T_Classe := PRIMAIRE) is
record
    --PARTIE FIXE
    francais, math, sport : float; --ou float range 0.0..20.0;
    --PARTIE VARIABLE : COLLEGE ET LYCEE
    case classe is
        when PRIMAIRE =>
            null;
        when COLLEGE | LYCEE =>
            lvl, bio : float;
    end case;
end record;
```

```

    lvl, bio : float ;
--PARTIE SPECIFIQUE AU LYCEE
  case classe is
    when LYCEE =>
      chimie, lv2 : float ;
    when others =>
      null ;
  end case ;
end record ;

```

Euh... Taurais pas oublier de faire une modification après ton copier-coller ? 😊

Non, regardez bien mon paramètre classe tout en haut : il a désormais une valeur par défaut : PRIMAIRE ! Vous allez me dire que ça ne change pas grand chose, et pourtant si ! Ainsi vous allez pouvoir écrire :

Code : Ada

```

Eric : T_Eleve ;          -- non prédefini
Bltn_Eric : T_Bulletin ;  -- pas la peine d'indiquer un paramètre, il est déjà prédefini

```

Et désormais, Bltn_Eric est un type mutant, il va pouvoir changer de structure **en cours d'algorithme** ! Nous pourrons ainsi écrire :

Code : Ada

```

...
Eric : T_Eleve ;
Bltn_Eric : T_Bulletin ;
BEGIN
  Bltn_Eric := (COLLEGE,12.5,13.0,9.5,15.0,8.0) ;
  Bltn_Eric := Bltn_Kevin ;           -- On suppose bien-sûr que Bltn_Kevin est déjà "prérempli"
...

```

Par contre, comme le fait d'attribuer une classe à Bltn_Eric va modifier toute sa structure, il est interdit de noter uniquement :

Code : Ada

```
Bltn_Eric.classe := COLLEGE ;
```

Toute changement de la composante classe doit se faire de manière «globale», c'est-à-dire «tout en une seule fois» !

Autre contrainte, si vous déclarez votre objet Bltn_Eric de la manière suivante :

Code : Ada

```
Bltn_Eric : T_Bulletin(COLLEGE) ;
```

Alors votre objet ne sera plus mutable ! On revient au cas d'un simple type structuré polymorphe. Donc ne spécifiez pas la classe durant la déclaration si vous souhaitez pouvoir changer la structure.

Toujours plus loin !

Pour aller plus loin il serait judicieux de «fusionner» les types T_Eleve et T_Bulletin de la manière suivante :

Code : Ada

```

type T_Bulletin(classe : T_Classe := PRIMAIRE) is
  -- !!! POUR L'INSTANT, PAS DE CHANGEMENTS !!!
  record
    --PARTIE FIXE
    français, math, sport : float ;   --ou float range 0.0..20.0 ;
    --PARTIE VARIABLE : COLLEGE ET LYCEE
    case classe is
      when PRIMAIRE =>
        null ;
      when COLLEGE | LYCEE =>
        lvl, bio : float ;
    end case ;
  end record ;
type T_Eleve is
  -- !!! LA MODIFICATION SE FAIT ICI !!!
  record
    nom : string(1..20) ;
    prenom : string(1..20) ;
    bulletin : T_Bulletin ;
  end record ;

```

Inutile de garder deux types : le type T_Bulletin sera intégré au type T_Eleve. En revanche, il faudra que le type T_Eleve soit déclaré après le type T_Bulletin. Si cela se révélait impossible, vous pouvez toujours éviter ce problème en écrivant la spécification de T_Eleve, puis en décrivant T_Bulletin et enfin T_Eleve :

Code : Ada

```

type T_Eleve ;           -- !!! AJOUT D'UNE
SPECIFICATION !!! Le reste ne change pas.

type T_Bulletin(classe : T_Classe := PRIMAIRE) is
record
  ...
end record ;

type T_Eleve is
record
  ...
end record ;

```

Plus besoin non plus de garder une composante classe au sein du type T_Eleve puisqu'elle existe déjà dans la composante bulletin !

Vous voilà désormais armés pour créer vos propres types (modulaires, énumérés, sous-types, structurés [non paramétrés, polymorphe ou mutant]). Combiné à l'usage des tableaux, voilà qui élargit vraiment nos horizons et nos possibilités. Si nous développions notre type T_Eleve et ses fonctionnalités qui s'y rapportent, nous pourrions créer un package volumineux et complet permettant d'établir une sorte de base de données des élèves d'un établissement : création et modification d'un «fichier élèves» enregistrant de nombreuses variables de type T_Eleve, le numéro de téléphone, l'adresse, les notes par matière (dans des tableaux inclus dans le type T_Eleve)... la seule limite est votre imagination et votre temps libre.

D'ailleurs, nous allons réaliser ce genre de programme dans le prochain chapitre : il s'agira de créer un logiciel gérant, non pas des

élèves, mais votre collection de CD, DVD... Comme vous l'avez deviné, notre prochain chapitre ne sera pas théorique, ce sera un TP ! Alors, si certains points vous posent encore problème, n'hésitez pas à les revoir.

En résumé :

- Un sous-type ou **SUBTYPE**, est une restriction d'un type préexistant. Par conséquent, les sous-types bénéficient de toutes les fonctionnalités offertes au type initial.
- Préférez créer un type énuméré plutôt que de représenter certaines propriétés par des nombres. Il est plus simple de comprendre qu'une personne est de sexe masculin que de comprendre que son sexe est le n°0.
- Les types articles ou structurés sont créées à l'aide du mot-clé **RECORD**. Contrairement aux tableaux, ils permettent de rassembler de nombreuses informations de types très divers dans des composantes. Ces composantes peuvent être elles-mêmes des types articles.
- Les types articles peuvent être paramétrés à la manière des fonctions. Ce paramétrage permet de créer des types polymorphes, c'est à dire qu'une partie de la structure du type dépendra du paramètre fourni. Lorsqu'une variable de type polymorphe est déclarée, son type doit absolument être paramétré.
- En fournissant une valeur par défaut au paramètre, vous pouvez définir un type mutant dont la structure pourra évoluer au cours de votre code. Plus aucun paramétrage n'est alors nécessaire à la déclaration de votre variable.

[TP] Logiciel de gestion de bibliothèque

Bienvenue dans le premier TP de la partie III, c'est-à-dire le second TP de ce cours. Comme je vous le disais dans le précédent chapitre, nous allons cette fois créer un logiciel pour gérer votre collection de CD, DVD, VHS, BluRay... Plus précisément, notre programme permettra d'enregistrer un film, un album de musique, un jeu vidéo ou un autre type de donnée, mais aussi de posteriori la liste des œuvres. Je vous propose d'appeler notre programme **Maktaba**, ce qui signifie «Bibliothèque» en Swahili (pourquoi pas ? On a bien Ubuntu, Amarok...).

Cela nous permettra de réutiliser les types de données structurés (pour enregistrer un film, il faut indiquer son titre, le type de support, son genre...), les types énumérés (les supports CD, DVD...), les fichiers binaires ou texte (pour enregistrer notre liste d'œuvres ou exporter des informations sur une œuvre), les strings (pour enregistrer les titres des morceaux de musique par exemple) ou encore les packages (notre programme devrait être assez conséquent).

Cette fois encore, je commencerai ce TP en vous fournitant un cahier des charges : que veut-on comme fonctionnalités ? Comme données ? Quelle structure pour nos fichiers et notre code source ? Puis, je vous guiderai dans la conception du programme : nous ne réaliserons pas tout en une fois, je commencerai par vous demander de réaliser un programme simple avant d'ajouter des fonctionnalités supplémentaires ou de prendre en charge des cas particuliers. Enfin, je vous transmettrai les sources et les spécifications d'une solution possible (bien entendu il n'y a pas qu'une seule solution mais plusieurs, chacune ayant ses avantages et inconvénients). En conclusion, comme pour le premier TP, je vous soumettrai quelques idées d'améliorations possibles de notre programme.

Prêt à démarer ce nouveau challenge ? Alors au travail ! 

Cahier des charges

Quelles données pour quels types de données ?

Pour établir les types dont nous pourrions avoir besoin, nous devons lister les données nécessaires à notre programme pour l'enregistrement ou la lecture.

Le contenu d'une œuvre

Nous souhaitons enregistrer des **œuvres**, mais qu'est-ce qu'une œuvre ? C'est avant tout :

- un **titre** : «sautant en l'air», «Mario bros 3», «Nevermind»...
- une **catégorie** : FILM, JEU (VIDÉO), ALBUM (DE MUSIQUE), AUTRE...
- un **support** : CD, DVD, BLURAY, VHS, HDDVD...
- une **note** : de zéro à trois étoiles.

Ensuite, selon la catégorie de l'œuvre, d'autres informations peuvent être nécessaires. Pour un film, nous aurons besoin :

- du nom du **réalisateur**.
- de savoir si le film est en **VF**.

Pour un jeu vidéo, nous aurons besoin :

- de la **console** : Nes, PS1, PC, Nintendo64...;
- de savoir si vous avez **terminé** le jeu.

Pour un album de musique, nous aurons besoin :

- du nom de l'**artiste** : «Nirvana», «ACDC», «Bob Marley»...
- D'une liste des **morceaux** dans l'ordre : «Come as you are», «Something in the way»...

Il serait bon également qu'un type d'œuvre par défaut existe.

Des types en cascade

Suite à cela, vous avez du comprendre que nous aurons besoin d'un type **T_Oeuvre** qui soit structuré et polymorphe (voire même mutable, ce serait encore mieux et je vous le conseille fortement). Ce type structuré devrait comprendre de nombreuses composantes de types aussi divers que du texte (string mais pas d'unbounded_string, je vous expliquerai plus tard pourquoi), des tableaux, des booléens, des types énumérés (pour les supports ou la catégorie)...

 **Rien que ça ? T'avais pas plus long ?** 

 C'est en effet beaucoup et en même temps bien peu ! Pensez que nous aurions pu enregistrer les durées des films ou des morceaux, si les films sont en VOSTFR, la pochette de l'album par exemple, l'emplacement où est sensé être rangé le CD (sur l'étagère, dans la tour, chez le voisin...) ou encore sa date d'achat ou de parution... Ayez en tête que la plupart des logiciels actuels sont bien plus complexes que cela, notre programme n'est que peu de chose à côté. Toutefois, il vous sera possible de le perfectionner plus tard.

Tout cela laisse supposer que nous devrions créer un package spécifique pour déclarer notre type **T_Oeuvre** afin de libérer notre code source principal. Enfin, dernière information, nos textes devant être enregistrés dans des fichiers, nous ne pourrons pas utiliser les unbounded_string, mais seulement les string. L'explication technique vous sera révélée à la fin de la partie III, lors du chapitre sur les listes. Mais cela implique donc que vos strings devront être suffisamment longs pour pouvoir accueillir des titres à rallonge comme celui-ci, extrait d'un album de Nirvana : «Frances Farmer will have her revenge on Seattle».

Les types de fichiers

Qui dit enregistrement, dit nécessairement fichiers. Aux vues de notre type structuré **T_Oeuvre**, cela signifie que nous aurions besoin d'un **fichier binaire** pour jouer le rôle de base de donnée (équivalent ou à accès direct, c'est à vous de voir). Pour ma part, j'ai choisi les fichiers séquentiels dont la manipulation sera plus simple pour vous). Il serait même bon de séparer les bases de données (une pour les jeux, une pour la musique...). Ces fichiers porteront par conséquent les noms de «ListeJeu.bdd», «ListeAlbumbdd», «ListeFilm.bdd» et «ListeAutre.bdd» (bdd = **Base De Données**).

Quelle architecture pour les fichiers

Notre programme ayant besoin de divers types de fichiers, il serait judicieux de ne pas tout mélanger.

- **Maktaba.exe** devra se trouver dans un répertoire Maktaba, libre à vous de placer un raccourci sur le bureau si vous le souhaitez.
- Les bases de données **ListeJeu.bdd** et autres devront se trouver dans un sous-répertoire «data».
- Un sous-répertoire «Manual» permettra l'enregistrement d'un fichier texte.

Cela nous fait donc un répertoire principal et deux sous-répertoires.

Quelles fonctionnalités pour quelles fonctions et procédures ?

Maktaba.exe devra proposer les fonctionnalités suivantes :

- **Saisie** d'une nouvelle œuvre par l'utilisateur.
- **Enregistrement** d'une nouvelle œuvre par l'utilisateur (ajout dans la base de donnée).
- **Modification** par l'utilisateur d'une œuvre existante.
- **Suppression** par l'utilisateur d'une œuvre existante.
- **Affichage** de la base de donnée ou d'une œuvre dans la console (sous la forme d'un tableau).
- Accès aux fonctionnalités par **lignes de commande** : l'utilisateur devra taper `add`, `modify`, `delete`, `print` pour accéder à une fonctionnalité. Le programme gèrera bien-sûr d'éventuelles erreurs de frappe commises par l'utilisateur.
- Possibilité d'accéder à un **manuel en console** par la commande `manual`. Ce manuel sera rédigé dans un fichier texte «manual.txt» placé dans le sous-répertoire «manual» évoqué précédemment et décrira les différentes commandes possibles.

Architecture du code source

 Argh !!! Mais jamais je ne parviendrai à faire tout ça ! Je savais bien que je n'aurais pas du me lancer dans cette galère ! 

Gardez espoir ! Ce ne sera pas aussi compliqué que cela peut paraître. En revanche, cela risque d'être long (notamment le codage des procédures de saisie et d'affichage), donc il sera nécessaire d'y aller étape par étape et d'adopter une approche par modules (tiens, ça devrait vous rappeler les packages ça). Nous aurons donc à créer les fichiers Ada suivants :

- **Maktaba.adb** : la procédure principale qui ne se chargera que de l'interface utilisateur, du cœur du logiciel.
- **Maktaba.Types.ads** pour déclarer nos types et nos constantes.
- **Maktaba_Functions.adb** et **Maktaba_Functions.ads** : pour les différentes fonctionnalités liées à la base de données (lecture, saisie, enregistrement, affichage).

Conception du programme (suivez le guide)

Cette partie n'est pas obligatoire, elle permettra toutefois à ceux qui hésitent à se lancer ou qui ne voient pas comment faire, de trouver une méthode ou des voies de programmation. Attention, il ne s'agit pas d'une solution toute faite (celle-ci sera fournie à la fin) mais plutôt d'un guide et je vous invite à essayer de réaliser ce TP par vous-même, en recourant le moins possible à cette partie.

Création des types

Nous allons commencer par créer deux fichiers : Maktaba.adb et Maktaba_Types.ads. Notre fichier Maktaba.adb ne contiendra pour l'instant pas grand chose :

Code : Ada

```
WITH Maktaba_Types ;
      USE Maktaba_Types ;
WITH Ada.Text_IO ;
      USE Ada.Text_IO ;

PROCEDURE Maktaba IS
  Oeuvre : T_Oeuvre ;
BEGIN
END Maktaba ;
```

Comme vous pouvez le constater il n'y a quasiment rien. Nous allons nous concentrer sur le fichier Maktaba_Types.ads et les différents types : nous devons créer un type structuré et polymorphe (et même mutable) T_Oeuvre. Ce type devra contenir différentes composantes :

- titre de l'œuvre, réalisateur, console et artiste seront des strings (avec les contraintes que cela implique en terme de taille), mais surtout pas des unbounded_string (cela poserait problème pour l'enregistrement). Il serait bon que les strings soient initialisés.
- Catégorie et support seront des types énumérés.
- VF et Terminé seront des boolean (vrai ou faux).
- Note sera un sous-type naturel de 0 à 3 ou Integer (voire modulaire).
- Morceaux sera un tableau de 30 strings (ou plus).

À vous donc de créer ce type T_Oeuvre ainsi que les types énumérés T_Categorie et T_Support et le sous-type T_Note. Pensez toutefois que, pour être polymorphe, T_Oeuvre doit dépendre de la catégorie de l'œuvre et que pour être mutable, cette catégorie doit être initialisée :

Code : Ada

```
type T_Oeuvre(categorie : T_Categorie := #Une_Valeur#) is
  record
    ...
    case categorie is
      when FILM => ...
    ...
  end case;
end record;
```

Affichage d'une œuvre

Il est temps de créer nos fichiers Maktaba_Functions.adb et Maktaba_Functions.ads ! Nous aurons besoin à l'avenir d'afficher l'intégralité de notre base de données, mais avant d'afficher 300 œuvres, nous devrions créer une fonction ou procédure qui en affiche une et une seule. Cela nous permettra d'avoir dès et déjà un programme Maktaba.exe opérationnel qui saisitrait une œuvre (arbitrairement pour l'instant) puis l'afficherait.

Code : Ada

```
procedure Affichage(oeuvre : T_Oeuvre);
```

Pensez à faire un affichage compact et clair : il y aura à terme des dizaines d'œuvres ! Pensez également à terminer l'affichage par un ou plusieurs new_line pour éviter les soucis d'affichage plus tard.

Saisie d'une œuvre

La première chose à faire pour la suite sera de créer une procédure ou une fonction de saisie d'une œuvre. Le sous-programme de saisie ne sera pas compliqué à mettre en œuvre mais sera long à rédiger car il devra prévoir toutes les composantes. Mais avant de vous lancer dans la saisie d'une œuvre, je vous conseille d'implémenter une fonction de saisie de string. Il existe bien get_line ou get, mais si vous souhaitez saisir un string de taille 20, il ne faut pas que l'utilisateur saisisse un texte de 35 ou 3 caractères. Or l'utilisateur final n'aura généralement aucune connaissance de ces contraintes, donc je conseille de commencer par là :

Code : Ada

```
function get_text(taille : natural) return string;
```

Autre indication pour simplifier votre code, il serait bon que votre fonction de saisie d'œuvre (appelons-la Saisie_Oeuvre) ne s'occupe pas de la saisie de la catégorie. Ce travail sera effectué par une fonction tierce (Saisie_Categorie) qui fournira à la première la catégorie à saisir. Cela simplifiera grandement votre travail et votre réflexion. De manière générale, une grande partie de vos fonctions et procédures devraient avoir la catégorie de l'œuvre en paramètre.

Code : Ada

```
Function Saisie_Categorie return T_Categorie;
Function Saisie_Oeuvre(Cat : T_Categorie) return T_Oeuvre;
```

Les saisies de strings se feront avec notre fonction get_text. En revanche, les saisies d'entiers devront gérer les cas où l'utilisateur entre une note supérieure à 3 :

Code : Français

```
TANT QUE choix>3
| Saisir(choix)
FIN DE BOUCLE
```

De même, pour saisir un booléen ou un type structuré, vous pourrez proposer à l'utilisateur un choix similaire à celui-ci :

Code : Console

```
Votre film est enregistré sur :
1. un CD
2. un DVD
3. une VHS
Votre film est-il en VF ? (O : oui / N : Non) _
```

Donc prévoyez les cas où l'utilisateur répondrait de travers pour limiter les plantages.

Gestion des fichiers

La plupart des opérations suivantes se feront uniquement sur les fichiers : sauvegarde dans la base de données (bdd), affichage d'une bdd, modification d'un élément d'une bdd, suppression d'un élément d'une bdd... Nous devrons créer un package pour

manipuler des fichiers binaires. Comme dit précédemment, je vous invite à utiliser les fichiers séquentiels plutôt qu'à accès direct pour éviter de rajouter de la difficulté (bien sûr vous êtes libres de votre choix, le type de fichier binaire ne fait pas partie du cahier des charges).

Sauvegarde et affichage avec la BDD

L'implémentation de ces deux fonctionnalités ne devrait pas poser de problème. Il vous suffira d'ouvrir un fichier, de le fermer, en pensant entre temps à soit ajouter un élément (Append_File), soit parcourir le fichier pour lire (In_File).

Mais si vous avez essayé d'implémenter ces fonctionnalités, vous avez du vous rendre compte qu'elles exigent toutes les deux de commencer par traiter une question toute bête : «*Quel fichier dois-je ouvrir ?*». Et cette question, vous devrez vous la poser à chaque fois. Il y a donc plusieurs façons de faire : soit sur la joie gros boursin et alors «*Five le copier-coller !*», soit on est un peu plus tû et on redige une procédure qui se charge d'ouvrir le bon fichier selon la catégorie fournie en paramètre. Ce paramètre peut être lui-même fourni par la fonction Saisie_catégorie évoquée précédemment.

Code : Ada

```
procedure Ouvrir(cat : T_Categorie) ;
procedure Affichage_BDD(cat : T_Categorie) ;
procedure Sauvegarde(Oeuvre : T_Oeuvre) ;
```

Modification et suppression d'un élément de la BDD

Le plus simple pour effectuer cette opération est de ne manipuler que des fichiers : pas la peine de se casser la tête à tenter de supprimer un élément du fichier. Voici une méthode pour supprimer l'élément numéro N d'un fichier F :

Code : Français

```
Ouvrir le fichier F (Nom : truc)
Ouvrir le fichier G (Nom : truc2)
Copier les (N-1) premiers éléments de F dans G
sauter le N-ème élément de F
Copier le reste des éléments de F dans G
Supprimer F
Recréer F (Nom : Truc) comme une copie de G
Fermier F
Fermier G
```

Un raisonnement similaire peut être effectué pour la modification de la BDD.

Affichage du manuel

Là, j'espère bien que vous n'avez pas besoin de moi !

Les commandes

Jusque là, notre fichier Maktaba.adb ne contient rien de bien sérieux, il ne nous sert qu'à tester nos procédures et fonctions. Mais puisque nous avons fini, nous allons pouvoir le rédiger correctement. L'idée ici est simple : si l'utilisateur tape un mot particulier, le programme réalise une opération particulière. Nous allons donc réutiliser notre fonction get_text pour la saisie des commandes. Le corps de la procédure principale sera simple : une boucle infinie qui se contente de demander de saisir du texte et qui, si le texte correspond à une commande connue, lance quelques sous-programmes déjà rédigés. L'un de ceux-ci affichera bien entendu la sortie de la boucle (commande : *quit* ou *exit* par exemple). En cas d'erreur de l'utilisateur, le programme affichera toutefois une phrase au genre «*Si vous ne comprenez rien, vous n'avez qu'à taper Manuel pour lire ce #*\$%\$# de Manuel (RTFM)*» (en plus amiable bien sûr) de façon à ne pas laisser l'utilisateur sans indications.

Solutions possibles

Comme promis, voici une solution possible à comparer avec votre travail.

Maktaba.adb :

Secret (cliquez pour afficher)

Code : Ada

```
-----
-- PROJET MAKTABA --
-- 
-- AUTEUR : KAJI9 --
-- DATE : 13/11/2011 --
-- 
-- Contient la procédure principale du logiciel MAKTBA ainsi que
les --
-- procédures d'affichage et de saisie. --
-- 
-----
```

--Retrouvez le tuto à l'adresse suivante
<http://www.siteduzero.com/tutoriel-3-558031-1-second-tp-un-logiciel-de-gestion-de-bibliotheque.html>

```
with Maktaba_Types ;
with Maktaba_Functions ;
use Maktaba_Types ;
use Maktaba_Functions
;
with Ada.Characters.Handling ;
Ada.Characters.Handling ;
with Ada.Text_IO ;
use Ada.Text_IO ;

procedure Maktaba is
oeuvre : T_Oeuvre ;
reponse : string(1..6) ;
begin
loop
Put("> ") ; réponse := Get_Text(6) ;
réponse := To_Upper(Réponse) ;
if réponse = "QUIT " or réponse = "EXIT "
then exit ;
elsif réponse = "NEW "
then oeuvre := saisie(get_catégorie) ;
sauvegarde(oeuvre) ;
elsif réponse = "MANUAL"
then Affichage_Manuel ;
elsif réponse = "INIT " or réponse = "ERASE "
then creation(get_catégorie) ;
elsif réponse = "PRINT "
then affichage_bdd(get_catégorie) ;
elsif réponse = "EDIT "
then Edit_bdd(get_catégorie) ;
else put_line("Commande inconnue. Pour plus
d'informations, tapez l'instruction MANUAL. ");
end if ;
end loop ;
end Maktaba ;
```

Maktaba_Types.ads :

Secret (cliquez pour afficher)

Code : Ada

```
-----
-- PROJET MAKTABA --
-- 
-- MAKTABA_TYPES.ADS --
-- 
-- 
-- AUTEUR : KAJI9 --
-- DATE : 13/11/2011 --
-- 
```

```
-- Contient les différents types nécessaires au fonctionnement du
logiciel --
-- MAKTABA. --
-----
-----
```

--Retrouvez le tuto à l'adresse suivante
<http://www.siteduzero.com/tutoriel-3-558031-1-second-tp-un-logiciel-de-gestion-de-bibliotheque.html>

```
package Maktaba_Types is
    type T_Categorie is (FILM,JEU,ALBUM,AUTRE) ;
    type T_Support is (CD, DVD, BLURAY, VHS, HHDVD) ;
    type T_Morceaux is array(1..30) of string(1..50) ;
    type T_Oeuvre(categorie : T_Categorie := AUTRE) is
        record
            titre : string(1..50) := (others => ' ') ;
            support : T_Support := CD ;
            note : natural range 0..3 ;
            case categorie is
                when FILM => realisateur : string(1..20) := (others => ' ')
                when JEU => VF : boolean := false ;
                when ALBUM => terminé : boolean := false ;
                when AUTRE => artiste : string(1..20) := (others => ' ')
            end case ;
            morceaux : T_Morceaux := (others =>(others
=> ' ')) ;
        end record ;
    type T_Film is new T_Oeuvre(FILM) ;
    type T_Jeu is new T_Oeuvre(JEU) ;
    type T_Album is new T_Oeuvre(ALBUM) ;
    type T_Autre is new T_Oeuvre(AUTRE) ;
end ;
```

Maktaba_Functions.ads :

Secret (cliquez pour afficher)

Code : Ada

```
-----
-- PROJET MAKTABA --
-- MAKTABA_FUNCTIONS.ADS --
-- --
-- AUTEUR : KAJI9 --
-- DATE : 13/11/2011 --
-- --
-- Contient les fonctions de saisie, d'affichage, de sauvegarde...
du
-- logiciel MAKTABA. --
-----
```

--Retrouvez le tuto à l'adresse suivante
<http://www.siteduzero.com/tutoriel-3-558031-1-second-tp-un-logiciel-de-gestion-de-bibliotheque.html>

```
with Maktaba_Types ;
WITH Ada.Sequential_IO ;
use Maktaba_Types ;

package Maktaba_Functions is
    Package_P_Fichier is new Ada.Sequential_IO(T_Oeuvre) ;
    use P_Fichier ;
    subtype T_Fichier is P_Fichier.File_type ;
    procedure sauvegarde(Oeuvre : in T_Oeuvre) ;
    procedure sauvegarde(oeuvre : in T_Oeuvre ; rang : natural) ;
    procedure creation(cat : T_Categorie) ;
    procedure Ouvrir(F: in out T_Fichier ; mode :
P_Fichier.File_Mode := P_Fichier.In_file ; cat : T_Categorie);
    procedure supprimer(cat : T_Categorie ; rang : natural) ;
    -- SAISIES
    function get_text(size : integer) return string ;
    function get_categorie return T_Categorie ;
    function saisie(cat : T_Categorie) return T_Oeuvre ;
    -- MANIPULATION DE LA BDD
    procedure affichage_oeuvre(Oeuvre : in T_Oeuvre) ;
    procedure affichage_bdd(cat : T_Categorie) ;
    procedure Edit_bdd(cat : T_Categorie) ;
    -- Manuel
    procedure affichage_manuel ;
end Maktaba_Functions ;
```

Maktaba_Functions.adb :

Secret (cliquez pour afficher)

Code : Ada

```
-----
-- PROJET MAKTABA --
-- MAKTABA_FUNCTIONS.ADB --
-- --
-- AUTEUR : KAJI9 --
-- DATE : 13/11/2011 --
-- --
-- Contient les fonctions de saisie, d'affichage, de sauvegarde
... du
-- logiciel MAKTABA. --
-----
```

--Retrouvez le tuto à l'adresse suivante
<http://www.siteduzero.com/tutoriel-3-558031-1-second-tp-un-logiciel-de-gestion-de-bibliotheque.html>

```

with Ada.Text_IO ;           use Ada.Text_IO ;
with Ada.Integer_Text_IO ;    use Ada.Integer_Text_IO ;
Ada.Characters.Handling ;   use Ada.Characters.Handling ;
Ada.Characters.Handling ;   use Ada.Characters.Handling ;
with ada.Strings.Unbounded; use ada.Strings.Unbounded ;

package body Maktaba_Functions is

  -- GESTION DES FICHIERS --

procedure sauvegarde(Oeuvre : in T_Oeuvre) is
  F : T_Fichier ;
begin
  ouvrir(F,Append_File,oeuvre.categorie) ;
  write(F,Oeuvre) ;
  close(F) ;
end sauvegarde ;

procedure sauvegarde(oeuvre : in T_Oeuvre ; rang : natural) is
  F,G : T_Fichier ;
  tmp : T_Oeuvre ;
begin
  -- Ouverture de F en In et de G en Append
  ouvrir(F,In_File,oeuvre.categorie) ;
  create(G,Append_File,Name(F) & "2") ;
  -- copie de F oeuvre dans G
  for i in 1..rang-1 loop
    read(F,tmp) ;
    write(G,tmp) ;
  end loop ;
  write(G,oeuvre) ;
  read(F,tmp) ; -- lecture de l'élément à supprimer
  while not end_of_file(F) loop
    read(F,tmp) ;
    write(G,tmp) ;
  end loop ;
  -- Suppression de F / recréation de F
  -- fermeture de G / réouverture de G
  delete(F) ;
  creation(oeuvre.categorie) ;
  ouvrir(F,Append_File,oeuvre.categorie) ;
  close(G) ;
  open(G,In_File,Name(F) & "2") ;
  -- copie de G dans F
  while not end_of_file(G) loop
    read(G,tmp) ;
    write(F,tmp) ;
  end loop ;
  -- Fermeture de F / Suppression de G
  close(F) ;
  delete(G) ;
end sauvegarde ;

procedure Creation(cat : T_Categorie) is
  F : T_Fichier ;
begin
  case cat is
    when FILM => create(F,out_file,"./data/ListeFilm.bdd") ;
    when JEU => create(F,out_file,"./data/ListeJeu.bdd") ;
    when ALBUM => create(F,out_file,"./data/ListeAlbum.bdd") ;
    when AUTRE => create(F,out_file,"./data/ListeAutre.bdd") ;
  end case ;
end creation ;

procedure Ouvrir(F: in out T_Fichier ;
                 mode : P_Fichier.File_Mode := P_Fichier.In_file ;
                 cat : T_Categorie) is
begin
  case cat is
    when FILM => open(F,mode,"./data/ListeFilm.bdd") ;
    when JEU => open(F,mode,"./data/ListeJeu.bdd") ;
    when ALBUM => open(F,mode,"./data/ListeAlbum.bdd") ;
    when AUTRE => open(F,mode,"./data/ListeAutre.bdd") ;
  end case ;
end Ouvrir ;

procedure supprimer(cat : T_Categorie ; rang : natural) is
  F,G : T_Fichier ;
  tmp : T_Oeuvre ;
begin
  -- Ouverture de F en In et de G en Append
  ouvrir(F,In_File,cat) ;
  create(G,Append_File,Name(F) & "2") ;
  -- copie de F-1 oeuvre dans G
  for i in 1..rang-1 loop
    read(F,tmp) ;
    write(G,tmp) ;
  end loop ;
  read(F,tmp) ;
  while not end_of_file(F) loop
    read(F,tmp) ;
    write(G,tmp) ;
  end loop ;
  -- Suppression de F / recréation de F
  -- fermeture de G / réouverture de G
  delete(F) ;
  creation(cat) ;
  ouvrir(F,Append_File,cat) ;
  close(G) ;
  open(G,In_File,Name(F) & "2") ;
  -- copie de G dans F
  while not end_of_file(G) loop
    read(G,tmp) ;
    write(F,tmp) ;
  end loop ;
  -- Fermeture de F / Suppression de G
  close(F) ;
  delete(G) ;
end supprimer ;

  -- SAISIE --
  -- SAISIE --

function get_text(size : integer) return string is
  U : Unbounded_String := Null_Unbounded_String ;
  T : string(1..size) := (others => ' ') ;
begin
  U := to_unbounded_string(get_line) ;
  if length(U) > size then
    T := substring(U)(1..size) ;
  else
    T := string(T(1..length(U))) := to_string(U) ;
    for i in length(U)+1..size loop
      T(i) := ' ' ;
    end loop ;
  end if ;
  return T ;
end get_text ;

function get_categorie return T_Categorie is
  choix_cat : character ;
begin
  Put_Line("Choisissez la categorie desiree : ") ;
  Put_Line("F-Film M-Album de Musique") ;
  Put_Line("J-Jeu Video Autre ?") ;
  Get_Immediate(choix_cat) ; choix_cat := to_upper(choix_cat) ;
  case choix_cat is
    when 'F' => return FILM ;
  end case ;
end get_categorie ;

```

```

        when 'g' => return ALBUM ;
        when 'j' => return JEU;
        when others => return AUTRE ;
    end case ;
end get_categorie ;

function saisie(cat : T_Categorie) return T_Oeuvre is
    oeuvre : T_Oeuvre(cat);
    choix : character;
    note : integer;
begin
    -- SAISIE DES PARAMETRES COMMUNS
    Put_line("Quel est le titre de l'oeuvre ? ");
    oeuvre.titre := get_text(50);
    Put_line("Quelle note donneriez-vous ? (Entre 0 et 3) ");
    loop
        get(note); skip_line;
        if note in 0..3
            then oeuvre.note := note;
            exit;
        else Put_line("ERREUR ! La note doit être comprise
entre 0 et 3 !");
            end if;
        end loop;

    Put_line("Sur quel type de support l'oeuvre est-elle
enregistrée ?");
    Put_line("1-VHS 2-CD 3-DVD");
    Put_line("4-HDDVD 5-BLURAY");
    loop
        get immediate(choix); choix := to_upper(choix);
        case choix is
            when '1' => oeuvre.support := VHS; exit;
            when '2' => oeuvre.support := CD; exit;
            when '3' => oeuvre.support := DVD; exit;
            when '4' => oeuvre.support := HDDVD; exit;
            when '5' => oeuvre.support := BLURAY; exit;
            when others => Put_line("Veuillez reconfirmer votre
choix.");
        end case;
    end loop;
    -- SAISIE DES PARAMETRES SPECIFIQUES
    case cat is
        when FILM => Put_line("Quel est le réalisateur ? ");
        oeuvre.realisateur := get_text(20);
        Put_line("Le film est-il en VF ? (O : Oui / N
: Non)");
        loop
            get_immediate(choix); choix :=
to_upper(choix);
            if choix = 'O'
                then oeuvre.vf := true; exit;
            elsif choix = 'N'
                then oeuvre.vf := false; exit;
            else Put_line("Veuillez appuyer sur O
pour Oui ou sur N pour Non");
                end if;
            end loop;
            for i in oeuvre.morceaux'range loop
                Put_line("Voulez-vous ajouter un morceau ?
(O : Oui / N : Non)");
                get_immediate(choix); choix :=
to_upper(choix);
                if choix = 'O'
                    then Put_line("Quel est le titre du
morceau ? ");
                    oeuvre.morceaux(i) := get_text(50);
                else exit;
                end if;
            end loop;
        return oeuvre;
    when JEU => Put_line("Quelle est la console ? ");
        oeuvre.console := get_text(20);
        Put_line("Avez-vous fini le jeu ? (O : Oui /
N : Non)");
        loop
            get_immediate(choix); choix :=
to_upper(choix);
            if choix = 'O'
                then oeuvre.terminé := true; exit;
            elsif choix = 'N'
                then oeuvre.terminé := false; exit;
            else Put_line("Veuillez appuyer sur O
pour Oui ou sur N pour Non");
                end if;
            end loop;
            return oeuvre;
        when AUTRE => return oeuvre;
    end case;
end Saisie;

----- AFFICHAGE D'UNE BDD -----
-----
```

```

procedure affichage_oeuvre(oeuvre : T_oeuvre) is
    --Affiche une seule oeuvre
    null_string : constant string(1..50) := (others => ' ');
begin
    put(" >>>Titre : "); put(Oeuvre.titre); new_line;
    put(" >>>Support : "); put(T_Support'image(Oeuvre.support));
    new_line;
    put(" >>>Note : "); put(Oeuvre.note); new_line;
    case oeuvre.categorie is
        when FILM => put(" >>>Réalisateur : ");
        put(Oeuvre.realisateur); new_line;
        put(" >>>VF : ");
        if Oeuvre.vf
            then put("oui");
            else put("non");
        end if; new_line(2);
        when JEU => put(" >>>Console : "); put(Oeuvre.console);
        new_line;
        put(" >>>Terminé : ");
        if Oeuvre.terminé
            then put("oui");
            else put("non");
        end if; new_line(2);
        when ALBUM => put(" >>>Artiste : "); put(Oeuvre.artiste);
        new_line;
        put(" >>>Morceaux : ");
        for i in Oeuvre.morceaux'range loop
            exit when Oeuvre.morceaux(i) = null_string;
            put(" "); put(i,2); put(" : ");
        Put(oeuvre.morceaux(I)); New_line;
        end loop;
        new_line(2);
        when AUTRE => new_line;
    end case;
end affichage_oeuvre;
```

```

procedure affichage_bdd(cat : T_categorie) is
    --Affiche toute une base de données selon la catégorie demandée
    n : natural := 1;
    suite : character;
    Oeuvre : T_Oeuvre(cat);
    F : T_Fichier;
begin
    ouvrir(F,in_File,cat);
    while not end_of_file(F) loop
        if n = 0
            then put("Cliquez pour continuer");
            get_immediate(suite); new_line;
        end if;
        read(F,Oeuvre);
        Affichage_oeuvre(Oeuvre);
        n := (n + 1) mod 5;
    end loop;
    close(F);
end Affichage_bdd;
```

```

---                                     -----
-- EDITION DE LA BASE DE DONNEES --
---

procedure Edit_bdd(cat : T_Categorie) is
  --Edite une base de données pour modification ou suppression
  choix : character ;
  n : natural := 1 ;
  Oeuvre : T_Oeuvre(cat) ;
  F : T_Fichier ;
begin
  ouvrir(F,In_File,cat) ;
  while not end_of_file(F) loop
    read(F,Oeuvre) ;
    Affichage_oeuvre(oeuvre) ;
    put_line("Que voulez-vous faire : Supprimer(S), Modifier(M),
    Quitter(Q) ou Continuer ?") ;
    Get_Immediate(choix) ; choix := to_upper(choix) ;
    if choix = 'M'
      then close(F) ;
      oeuvre := saisie(cat) ;
      sauvegarde(oeuvre,n) ;
      exit ;
    elsif choix = 'S'
      then close(F) ;
      Supprimer(cat,n) ;
      exit ;
    elsif choix = 'Q'
      then close(F) ;
      exit ;
    end if ;
    n := n + 1 ;
  end loop ;
  if Is_Open(F)
    then close(F) ;
  end if ;
end Edit_bdd ;

---                                     -----
-- EDITION DE LA BASE DE DONNEES --
---

procedure affichage_manuel is
  F : Ada.Text_IO.File_Type ;
begin
  open(F,In_File,"./manual/manual.txt") ;
  while not end_of_file(F) loop
    put_line(get_line(F)) ;
  end loop ;
  close(F) ;
end affichage_manuel ;
end Maktaba_Functions ;

```

Manual.txt :

[Secret \(cliquez pour afficher\)](#)

Code : Français

Commandes :	
Quitter	: Quit / Exit
Nouveau	: New
Manuel	: Manual
Réinitialiser une base	: Init / Erase
Afficher une base	: Print
Modifier une base	: Edit

Pistes d'amélioration :

- Fournir une interface graphique plutôt que cette vilaine console. Pour l'instant, vous ne pouvez pas encore le faire, mais cela viendra.
- Proposer de gérer d'autres types de supports : disque durs, clés USB...
- Proposer le enregistrement de nouvelles informations : le genre de l'œuvre (film d'horreur ou d'aventure, album rock ou rap...), une description, les acteurs du film, les membres du groupe de musique, l'édition du jeu vidéo, la durée des morceaux...
- Proposer l'exportation de vos bases de données sous formes de fichiers textes consultables plus aisément ou l'importation d'une œuvre à partir d'un fichier texte.
- Permettre de compléter les instructions en ligne de commande par des options : *modify -f* modifierait un film par exemple.
- Implémenter un service de recherche d'œuvre par titre, par auteur ou même par mots-clés !

Encore une fois, ce ne sont pas les idées qui manquent pour développer un tel logiciel. J'espère toutefois que ce TP vous aura permis de faire un point sur les notions abordées depuis le début de la partie III car nous allons maintenant nous atteler à un type de données plus complexe : les pointeurs. Et, de la même manière que nous avons parlé, reparté et rereparlé des tableaux, nous allons parler, repartir et rerepartir des pointeurs dans les prochains chapitres.

Les pointeurs I : allocation dynamique

Voici très certainement LE chapitre de la troisième partie du cours. Pourquoi ? Eh bien parce que c'est, à n'en pas douter le plus dur de tous. Les pointeurs constituent une véritable épine dans le pied de tout jeune programmeur. Non pas qu'ils soient compliqués à manipuler, non bien au contraire ! Vous connaissez déjà les opérations qui leur sont applicables. Non. Le souci, c'est qu'ils sont difficiles à conceptualiser, à comprendre.

Comment ça marche ? Je pointe sur quoi en ce moment ? À quoi ça sert ? C'est un pointeur ou un objet pointé ? Je mets un pointeur ou pas ? Au secours ! 🤪 Voilà résumé en quelques questions tout le désarroi qui vous submergera sûrement en manipulant les pointeurs. C'est en tous cas celui qui m'a submergé lorsque j'ai commencé à les utiliser et ce, jusqu'à obtenir le déclic. Et, assurez-vous, je compte bien prendre le temps de vous expliquer pour que vous aussi vous ayiez le déclic à votre tour.

Ce chapitre étant long et compliqué il a donc été divisé en deux. La première moitié (celle que vous lisez actuellement) se chargera de vous expliquer les bases : qu'est-ce qu'un pointeur ? Comment cela marche-t-il ? À quoi cela ressemble-t-il en Ada ? La seconde moitié aura pour but de couvrir toutes les possibilités offertes par le langage Ada en la matière. Aussi compliqués soient-ils (et même si le langage Ada fait moins appel à eux que d'autres comme le C/C++), les pointeurs nous seront très utiles par la suite notamment pour le chapitre 10 de la partie V. N'hésitez pas à relire ce double chapitre, si besoin est, pour vous assurer que vous avez assimilé toutes les notions.

Mémoire, variable et pointeur

Notre premier objectif sera donc de comprendre ce qu'est un pointeur. Et pour cela, il faut avant tout se rappeler ce qu'est une variable !

On va pas tout reprendre à zéro quand même ? 🤪

Et si ! Ou presque. Reprenons calmement et posons-nous la question suivante : que se passe-t-il lorsque l'on écrit dans la partie déclarative la ligne ci-dessous et à quoi cela sert-il ?

Code : Ada

```
MaVariablePerso : integer;
```

En écrivant cela, nous indiquons à l'ordinateur que nous aurons besoin d'un espace en mémoire suffisamment grand pour accueillir un nombre de type integer, c'est-à-dire compris entre - 2 147 483 648 et + 2 147 483 647. Mon ordinateur va devoir réquisitionner une zone mémoire pour ma variable MaVariablePerso. Et pour la retrouver, cette zone mémoire disposera d'un adresse, par exemple le n°6025. Ainsi, à cette adresse, je suis sûr de trouver MaVariablePerso et rien d'autre ! Il ne peut y avoir qu'une seule variable par adresse.

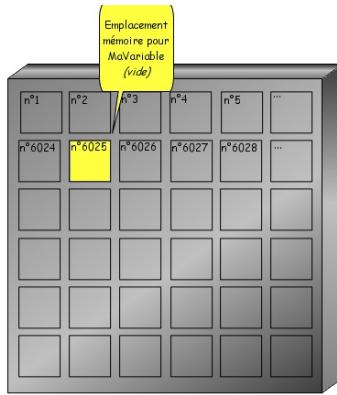


Schéma de la mémoire après

déclaration d'une variable

Ainsi, si par la suite je viens à écrire « MaVariablePerso := 5 ; », alors l'emplacement mémoire n°6025 qui était encore vide (ou qui contenait d'anciennes données sans aucun rapport avec notre programme actuel) recevra la valeur 5 qu'il conservera jusqu'à la fin du programme.

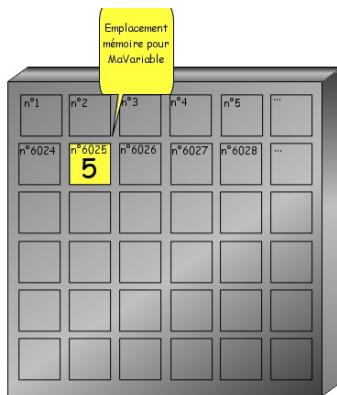


Schéma de la mémoire après

affectation d'une valeur à notre variable

En revanche, nous ne pourrons pas écrire « MaVariablePerso := 5.37 ; » car 5.37 est un float, et un float n'est pas codé de la même manière que l'ordinateur qu'un integer ou un caractère. Donc l'ordinateur sait qu'à l'adresse n°6025, il ne pourra enregistrer que des données de type integer sous peine de plantage. L'avantage des langages dits de haut niveau (tels Ada), c'est qu'ils s'occupent de gérer la réquisition de mémoire, éviter en amont de mauvaises affectations et surtout, ces langages nous permettent de ne pas avoir à nous tracasser de l'adresse mémoire qu'il n°6025. Il nous suffit d'indiquer le nom de cet emplacement (MaVariablePerso) pour pouvoir y accéder en écriture ou/et en lecture.

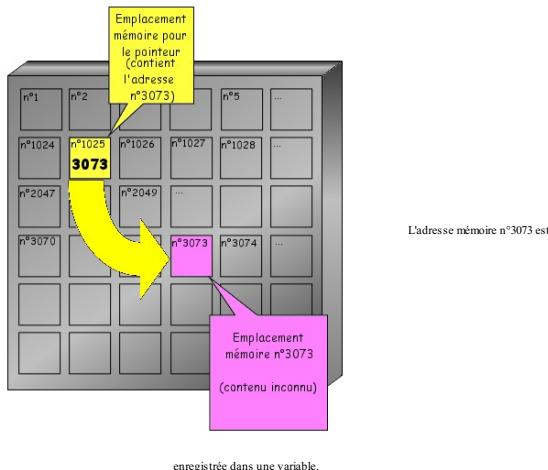
Or, il est parfois utile de passer non plus par le nom de cet emplacement, mais par son adresse, et c'est là qu'interviennent les **pointeurs** aussi appelés **accessseurs** en Ada ! Au lieu de dire « je veux modifier/lire la variable qui s'appelle MaVariablePerso », nous devrons dire « je veux modifier/lire ce qui est écrit à l'adresse n°6025 ».

Pour comprendre ce qu'est un pointeur, mettons-nous dans la peau de l'inspecteur Derrick (la classe ! Non ?). Il sait que le meurtrier qu'il recherche vit à l'adresse 15 rue Mozart, bâtiment C, 5ème étage appartenant 34. Mais il ne connaît pas son nom. Que fait-il ? Soit il imprime quelque part dans son cerveau l'adresse du criminel (vue l'adresse, ça sera pas facile, mais... c'est Derrick ! 😊), soit il utilise un bon vieux post-it pour noter (moins glamour mais plus sérieux son âge). Ensuite, il se rend à l'adresse indiquée, fait ce qu'il a à faire (interrogatoire, arrestation...) termine l'enquête et youpi, il peut jeter son post-it car il a résolu une nouvelle enquête.

Et c'est quoi le pointeur dans cette histoire ? 😊

Eh bien mon pointeur c'est : le post-it ! Ou le coin du cerveau de Derrick qui a enregistré cette adresse si compliquée ! Traduit en langage informatique, cela signifie qu'une adresse mémoire est trop compliquée pour être retenue ou utilisée par l'utilisateur, mieux vaut l'enregistrer quelque part, et le plus sérieux, c'est d'enregistrer cette adresse dans une bonne vieille variable tout ce

qu'il y a de plus classique (ou presque). Et c'est cette variable, contenant une adresse mémoire intéressante qui est notre pointeur.



Sur mon schéma, le pointeur est donc la case n°1025 ! Cette case contient l'adresse (le n°3073) d'un autre emplacement mémoire. Un pointeur n'est donc rien de plus qu'une variable au contenu inutilisable en tant que tel puisque ce contenu n'est rien d'autre que l'adresse mémoire d'informations plus importantes. Le pointeur n'est donc pas tant intéressant par ce qu'il contient que par ce qu'il pointe.

i Nous verrons par la suite que c'est tout de même un poil plus compliqué.

Le type access

Déclarer un pointeur

Le terme Ada pour pointeur est **ACCESS** ! C'est pourquoi on parle également d'accesseur. Malheureusement, comme pour les tableaux, il n'est pas possible d'écrire :

Code : Ada

```
Ptr : access ;
```

Toujours pour éviter des problèmes de typage, il nous faut définir auparavant un type **T_Pointeur** indiquant sur quel type de données nous souhaitons pointer. Pour changer, choisissons que le type **T_Pointeur** pointera sur... des integer ! Oui je sais, c'est toujours des integer. 😊 Nous allons donc déclarer notre type de la manière suivante :

Code : Ada

```
type T_Pointeur is access Integer ;
```

Puis nous pourrons déclarer des variables **Ptr1**, **Ptr2** et **Ptr3** de type **T_Pointeur** :

Code : Ada

```
Ptr1, Ptr2, Ptr3 : T_Pointeur ;
```

Que contient mon pointeur ?

En faisant cette déclaration, le langage Ada initialise automatiquement vos pointeurs (tous les langages ne le font pas, il est alors important de l'initialiser soi-même). Mais attention ! Ils ne pointent sur rien pour l'instant (il ne faudrait pas qu'ils pointent au hasard sur des données de type float, caractère ou autre qui n'auraient pas été effacées). Ces pointeurs sont donc initialisés avec une valeur nulle : **NULL**. C'est comme si nous avions écrit :

Code : Ada

```
Ptr1, Ptr2, Ptr3 : T_Pointeur := Null ;
```

Autrement dit, ils sont vides !



Bon alors maintenant, comment on leur donne une valeur utile ?

Ooh ! Malheureux ! Vous oubliez quelque chose ! La valeur du pointeur n'est pas utile en tant que telle car elle doit correspondre à un adresse mémoire qui, elle, nous intéresse ! Or le souci, c'est que cet emplacement mémoire n'est pas encore créé. À quoi bon affecter une adresse à notre pointeur s'il n'y a rien à l'adresse indiquée ? La première chose à faire est de créer cet emplacement mémoire (et bien entendu, de le lire par la même occasion à notre pointeur). Pour cela nous devons écrire :

Code : Ada

```
Ptr1 := new Integer ;
```

Ainsi, **Ptr1** « pointera » sur une adresse qui elle, comportera un integer !



Où dois-je écrire cette ligne ? Dans la partie déclaration ?

Non, surtout pas ! Je sais que c'est normalement dans la partie déclarative que se font les réquisitions de mémoire, mais nous n'y déclarons que le type **T_Pointeur** et nos trois pointeurs **Ptr1**, **Ptr2** et **Ptr3**. La ligne de code précédente doit s'écrire après le **BEGIN**, dans le corps de votre procédure ou fonction. C'est ce que l'on appelle l'**allocation dynamique**. Ce procédé permet, pour simplifier, de *créer et supprimer des variables n'importe quand durant le programme*. Nous entrerons dans les détails durant l'avant-dernière sous-partie, rassurez-vous. Retenez simplement qu'à la déclaration, nos pointeurs ne pointent sur rien et que pour leur affecter une valeur, il faut créer un nouvel emplacement mémoire.

Une dernière chose, pas la peine d'écrire des **Put(Ptr1)** ou **Get(Ptr1)**, car je vous ai un petit peu menti au début de ce chapitre, le contenu de **Ptr1** n'est pas un simple integer (ce serait trop simple), l'adresse contenue dans notre pointeur est un peu plus compliquée et d'ailleurs, vous n'avez pas besoin de la connaître, la seule chose qui compte c'est qu'elle pointe bien sur une adresse valide.

Comment accéder aux données ?



Et mon pointeur, il pointe sur quoi ? 😊

Ben... pour l'instant pas grand chose à vrai dire puisque l'emplacement mémoire est vide. 😊 Bah oui, un pointeur ça pointe et puis c'est tout... ou presque ! L'emplacement mémoire pointé n'a pas de nom (ce n'est pas une variable), il n'est donc pas possible pour l'heure de le modifier ou de lire sa valeur. Alors comment faire ? Une première méthode serait de revoir l'affectation de notre pointeur ainsi :

Code : Ada

```
Ptr1 := new integer'(124);
```

Ainsi, Ptr1 pointera sur un emplacement mémoire contenant le nombre 124. Le souci, c'est qu'on ne va pas réquisitionner un nouvel emplacement mémoire à chaque fois alors que l'on pourrait réutiliser celui existant ! Voici donc une autre façon :

Code : Ada

```
Ptr2 := new integer;
Ptr2.all := 421;
```

Eh oui, je vous avais bien dit que les pointeurs étaient un poil plus compliqué. Ils comportent une « sorte de composante » comme les types structurés ! Sauf que **ALL** a beau ressembler à une composante, ce n'est pas une composante du pointeur, mais la valeur contenue dans l'emplacement mémoire pointé !

Il existe une troisième façon de procéder, si l'on souhaite avoir un pointeur sur une variable ou une constante déjà connue. Mais nous verrons cette méthode plus loin, à la fin de ce chapitre.

Opérations sur les pointeurs

Quelles opérations peut-on donc faire avec des pointeurs ? Eh bien je vous l'ai déjà dit : pas grand chose de plus que ce que vous savez déjà faire ! Voir même moins de choses. Je m'explique : vous pouvez tester l'égalité (ou la non égalité) de deux pointeurs, effectuer des affectations... et c'est tout. Oh, certes vous pouvez effectuer toutes les opérations que vous souhaitez sur **Ptr1.all** puisque ce n'est pas un pointeur mais un integer ! Mais sur Ptr1, (et je ne me répète pas pour le plaisir) vous n'avez le droit qu'au test d'égalité (et non-égalité) et à l'affectation ! Vous pouvez ainsi écrire :

Code : Ada

```
if Ptr1 = Ptr2
then ...
```

Ou :

Code : Ada

```
if Ptr1 /= Ptr2
then ...
```

Mais surtout pas :



```
if Ptr1 > Ptr2
then ...
```

Ça n'aurait aucun sens ! En revanche vous pouvez écrire « `Ptr1 := Ptr2;` ». Cela signifiera que Ptr1 pointera sur le même emplacement mémoire que Ptr2 ! Cela signifie aussi que l'emplacement qu'il pointait auparavant est perdu : il reste en mémoire, sauf qu'en a perdu son adresse ! Cet emplacement sera automatiquement supprimé à la fin de notre programme.

Ce que je vous dis là est une mauvaise pratique : si l'emplacement est perdu, il n'empêche qu'il prend de la place en mémoire pour rien. Il serait bon de le supprimer immédiatement pour libérer de la place en mémoire pour d'autres opérations.

Autre problème : créer deux pointeurs sur une même donnée (comme nous l'avons fait dans le dernier exemple) est à la fois inutile, ou plutôt redondant (on utilise deux pointeurs pour faire la même chose, c'est du gaspillage de mémoire) et surtout dangereux ! Via le premier pointeur, on peut modifier la donnée. Et, sans que le second n'ait été modifié, la valeur sur laquelle il pointera ne sera plus la même. Exemple :

Code : Ada

```
Ptr2.all := 5;           --Ptr2 pointe sur "5"
Ptr1 := Ptr2;           --
On modifie l'adresse contenue dans Ptr1 ! !
Ptr1.all := 9;          --Ouais ! Ptr1 pointe sur "9" maintenant !
--Sauf que du coup, Ptr2 aussi puisqu'ils
--pointent sur le même emplacement ! ! !
```

Donc, évitez ce genre d'opération autant que possible. Préférez plutôt :

Code : Ada

```
Ptr2.all := 5;           --Ptr2 pointe sur "5"
Ptr1.all := Ptr2.all;    --Ptr1 pointe toujours sur la même adresse
On a simplement modifié le contenu de cette adresse
Ptr1.all := 9;          --Maintenant Ptr1 pointe sur "9" !
--Mais comme Ptr1 et Ptr2 pointent sur
--deux adresses distinctes, il n'y a pas de problème.
```

Une erreur à éviter

Voici un code dangereux :

Code : Ada

```
if ptr /= null and ptr.all > 0
then ...
```

Pourquoi à votre avis ?

Eh bien parce que si ptr ne pointe sur rien (s'il vaut **NULL**), lorsque vous effectuez ce test, le programme testera également si **ptr.all** est positif (supérieur à 0). Or si Ptr ne pointe sur rien du tout alors **ptr.all** n'existe pas !!! Nous devrons donc prendre la précaution suivante (comme nous l'avions vu avec les tableaux) :

Code : Ada

```
if ptr /= null and then ptr.all > 0
then ...
```

Libération de la mémoire

Un programme (un peu) gourmand

Avant de commencer cette partie, je vais vous demander de copier et de compiler le code ci-dessous :

Code : Ada

```
with ada.text_io;      use ada.Text_IO;
procedure aaa is
  type T_Pointeur is access Integer;
  P : T_Pointeur;
  c : character;
begin
```

```

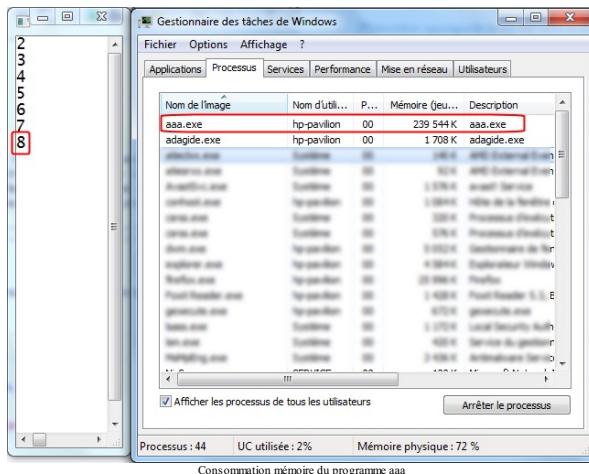
for i in 1..8 loop
    for j in 1..10**i loop
        P := new integer'(i*j);
    end loop;
    get(c); skip_line;
end loop;
end aaa;

```

Comme vous pouvez le voir, ce code ne fait rien de bien compliqué : il crée un pointeur auquel il affecte 10 valeurs successives, puis $10^2 = 100$, puis 1 000 ($10^3 = 1000$), puis 10 000 ($10^4 = 10000$... et ainsi de suite. Mais l'intérêt de ce programme n'est pas là. Nous allons regarder sa consommation mémoire. Pour cela, ouvrez un gestionnaire de tâche :

- si vous êtes sous Windows : appuyez sur les touches Alt+Ctrl+Suppr puis cliquez sur l'onglet Processus. Le programme apparaîtra en tête de liste.
- si vous êtes sous Linux ou Mac : un gestionnaire (appelé Moniteur d'activité sous Mac) est généralement disponible dans votre liste de programmes, mais vous pouvez également ouvrir une console et taper la commande `top`.

Lancez votre programme aaa.exe et admirez la quantité de mémoire utilisée : environ 780 Ko au démarrage mais après chaque saisie du caractère C, la mémoire utilisée augmente de manière exponentielle (quadratique en fait 😱). Voici une capture d'écran avec 200 Mo de mémoire utilisée par ce «petit» programme (mais on peut faire bien mieux).



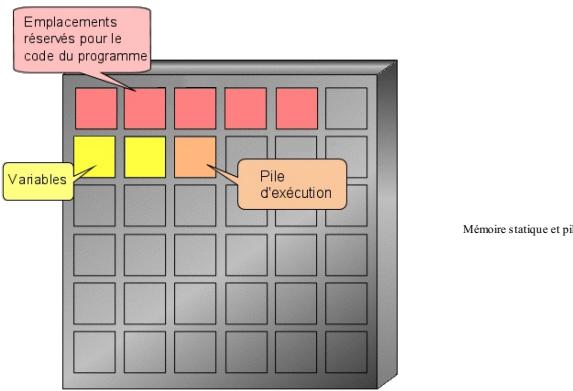
Mais comment se fait-il qu'un si petit programme puisse utiliser autant de mémoire ?

C'est ce que nous allons expliquer maintenant avant de fournir une solution à ce problème qui ne nous est jamais arrivé avec nos bonnes vieilles variables.

Un problème de mémoire

Reprenons nos schémas précédents car ils étaient un peu trop simplistes. Lorsque vous lancez votre programme, l'ordinateur va réserver des emplacements mémoire. Pour le code du programme tout d'abord, puis pour les variables que vous aurez déclaré entre `IS` et `BEGIN`. Ainsi, ayant même d'effectuer le moindre calcul ou affichage, votre programme va réquisitionner la mémoire dont il aura besoin : c'est à cela que sert cette fameuse partie déclarative et le typeage des variables. On dit que les variables sont stockées en **mémoire statique**, car la taille de cette mémoire ne va pas varier.

À cela, il faut ajouter un autre type de mémoire : la **mémoire automatique**. Au démarrage, l'ordinateur ne peut connaître les choix que l'utilisateur effectuera. Certains choix ne nécessiteront pas d'avantage de mémoire, d'autres entraîneront l'exécution de sous-programmes disposant de leurs propres variables ! Le même procédé de réservation aura alors lieu dans ce que l'on appelle la **pile d'exécution** : à noter que cette pile a cette fois une taille variable dépendant du déroulement du programme et des choix de l'utilisateur.



d'exécution

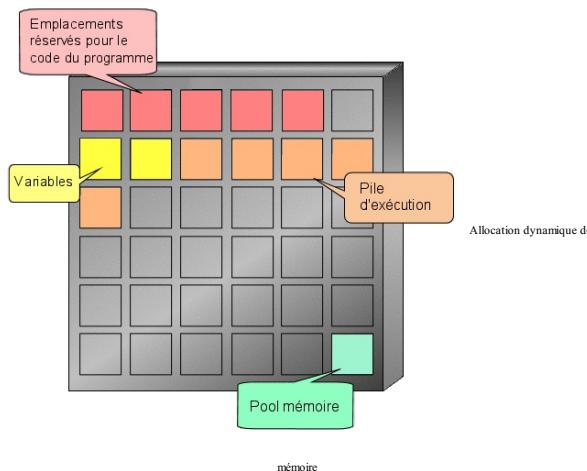
Mais ce procédé a un défaut : il gaspille la mémoire ! Toute variable déclarée ne disparaît qu'à la fin de l'exécution du programme ou sous-programme qui l'a générée. Ni votre système d'exploitation ni votre compilateur ne peuvent deviner si une variable est devenue inutile. Elle continue donc d'exister malgré tout. À l'inverse, ce procédé ne permet pas de déclarer des tableaux de longueur indéterminée : il faut connaître la taille avant même l'exécution du code source ! Contrariant 😱.

C'est pourquoi existe un dernier type de mémoire appelé **Pool ou Tas**. Ce pool, comme la pile d'exécution, n'a pas de taille pré définie. Ainsi, lorsque vous déclarez un pointeur `P` dans votre programme principal, l'ordinateur réserve suffisamment d'emplacements en mémoire statique pour pouvoir enregistrer une adresse. Puis, lorsque votre programme lit :

Code : Ada

```
P := new integer; -- suivie éventuellement d'une valeur
```

Celui-ci va automatiquement allouer, dans le Pool, suffisamment d'emplacements mémoire pour enregistrer un Integer. On parle alors d'**allocaiton dynamique**. Donc si nous répétons cette instruction (un peu comme nous l'avons fait dans le programme donné en début de sous-partie), l'ordinateur réservera plusieurs emplacements mémoire dans le Pool.



⚠ À noter qu'avec un seul pointeur (en mémoire statique) on peut réserver plusieurs adresses dans le pool (mémoire dynamique) !

Résolution du problème

Unchecked_Deallocation

Il est donc important lorsque vous manipulez des pointeurs de penser aux opérations effectuées en mémoire par l'ordinateur. Allouer dynamiquement de la mémoire est certes intéressant mais peut conduire à de gros gaspillages si l'on ne réfléchit pas à la portée de notre pointeur : tant que notre pointeur existe en mémoire statique, les emplacements alloués en mémoire dynamique demeurent et s'accumulent.

❓ La mémoire dynamique n'était pas sensée régler ce problème de gaspillage justement ?

Bien sûr que oui. Nous disposons d'une instruction d'allocation de mémoire (`NEW`), il nous faut donc utiliser une instruction de désallocation. Malheureusement celle-ci n'est pas aisée à mettre en œuvre et est risquée. Nous allons devoir faire appel à une procédure : `Ada.Unchecked_Deallocation`. Mais attention, il s'agit d'une procédure générique (faite pour tout type de pointeur) qui est donc inutilisable en tant que telle. Il va falloir lui créer une sœur jumelle, que nous appellerons `Free`, prévue pour notre type `T_Pointeur`. Voici le code corrigé :

Code : Ada

```
with ada.text_io ;      use ada.Text_IO ;
with Ada.Unchecked_Deallocation ;
```

```
procedure aaa is
  type T_Pointeur is access Integer ;
  P : T_Pointeur ;
  c : character ;
procedure free is new Ada.Unchecked_Deallocation(Integer,T_Pointeur)
;
begin
  for i in 1..8 loop
    for j in 1..10**i loop
      P := new integer'(i*j) ;
    free(P) ;
  end loop ;
  get(c) ; skip_line;
end loop ;
end aaa ;
```

Vous remarquerez tout d'abord qu'à la ligne 2, j'indique avec `WITH` que mon programme va utiliser la procédure `Ada.Unchecked_Deallocation` (pas de `USE`, cela n'aumit aucun sens puisque ma procédure est générique). Puis, à la ligne 8, je crée une procédure `Free` à partir de `Ada.Unchecked_Deallocation` ; on dit que l'on instance (mais nous verrons tout cela plus en détail dans la partie IV). Pour cela, il faut préciser tout d'abord le type de donnée qui est pointé puis le type de Pointeur utilisé.

⚠ Si nous avions deux types de pointeurs différents, il faudrait alors instancier deux procédures Free différentes.

Enfin, à la ligne 13, j'utilise ma procédure `Free` pour libérer la mémoire du Pool. Mon pointeur `P` ne pointe alors plus sur rien du tout et il est réinitialisé à `Null`. Vous pouvez compiler ce code et le tester : notre problème de «fuite de mémoire» a disparu !

Risques inhérents à la désallocation

Il est toutefois risqué d'utiliser `Ada.Unchecked_Deallocation` et la plupart du temps, je ne l'utiliserais pas. Imaginez que vous ayez deux pointeurs `P1` et `P2` pointant sur un même espace mémoire :

Code : Ada

```
...
P1 := new integer'(15) ;
P2 := P1 ;
free(P1) ;
Put(P2.all) ;
...
```

Le compilateur considéra ce code comme correct mais que se passera-t-il ? Lorsque `P1` est désalloué à la ligne 4, il est remis à `Null` mais cela signifie aussi que l'emplacement vers lequel il pointait disparaît. Et `P2` ? Il pointe donc désormais vers un emplacement mémoire qui a disparu et il ne vaut pas `Null` ! Voilà donc pourquoi on évitera la désallocation lorsque cela est possible.

Une autre méthode

❓ Je veux bien, mais tu nous exposes un problème, tu nous donne la solution et ensuite tu nous dis de ne surtout pas l'employer ! Je fais quoi moi ?

Une autre solution consiste à réfléchir (oui, je sais c'est dur 😊) à la portée de notre type `T_Pointeur`. En effet, lorsque le bloc dans lequel ce type est déclaré (la procédure ou la fonction le plus souvent) se termine, le type `T_Pointeur` disparaît et le pool de mémoire qu'il avait engendré disparaît lui aussi. Ainsi, pour éviter tout problème avec `Ada.Unchecked_Deallocation`, il peut être judicieux d'utiliser un bloc de déclaration avec l'instruction `DECLARE`.

Code : Ada

```
with ada.text_io ;      use ada.Text_IO ;
procedure aaa is
  c : character ;
begin
  for i in 1..8 loop
    for j in 1..10**i loop
  declare
    type T_Ptr is access Integer ;
    P : T_Ptr ;
```

```

begin
  p := new integer'(i*j) ;
end i;
  end loop ;
  get(c); skip_line;
end loop ;
end aaa ;

```

Exercices

Exercice 1

Énoncé

Créez un programme Test_Pointeur qui :

- crée trois pointeurs Ptr1, Ptr2 et Ptr3 sur des integer;
- demande à l'utilisateur de saisir les valeurs pointées par Ptr1 et Ptr2;
- enregistre dans l'emplacement mémoire pointé par Ptr3, la somme des deux valeurs saisies précédemment.

Je sais, c'est bidon comme exercice et vous pourriez le faire les yeux fermés avec des variables. Sauf qu'ici, je vous demande de manipuler non plus des variables, mais des pointeurs ! 😊

Solution

[Secret \(cliquez pour afficher\)](#)

Code : ada

```

WITH Ada.Text_IO; USE Ada.Text_IO;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Ada.Unchecked_Deallocation;

procedure Test_Pointeur is
  type T_Pointeur is access Integer;
  procedure Free is new Ada.Unchecked_Deallocation(Integer,
  T_Pointeur);
  Ptr1, Ptr2, Ptr3 : T_Pointeur;
begin
  Ptr1 := new Integer;
  Put("Quelle est la première valeur pointée ? ");
  get(Ptr1.all); skip_line;

  Ptr2 := new Integer;
  Put("Quelle est la seconde valeur pointée ? ");
  get(Ptr2.all); skip_line;

  Ptr3 := new Integer'(Ptr1.all + Ptr2.all); --on libère
  Free(Ptr1); --on libère
  Free(Ptr2); --on libère
  Put("Leur somme est de ");
  Put(Ptr3.all);
  Free(Ptr3); --on libère
  Put("Le troisième, mais est-ce vraiment utile ? ^");
end Test_Pointeur;

```

Exercice 2

Énoncé

Deuxième exercice, vous allez devoir créer un programme Inverse_Pointeur qui saisit une première valeur pointée par Ptr1, puis une seconde pointée par Ptr2 et qui inversera les pointeurs. Ainsi, à la fin du programme, Ptr1 devra pointer sur la seconde valeur et Ptr2 sur la première. Attention, les valeurs devront être saisies une et une seule fois, elles seront ensuite fixées !

Une indication : vous aurez sûrement besoin d'un troisième pointeur pour effectuer cet échange.

Solution

[Secret \(cliquez pour afficher\)](#)

Code : ada

```

WITH Ada.Text_IO; USE Ada.Text_IO;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
WITH Ada.Unchecked_Deallocation;

procedure Inverse_Pointeur is
  type T_Pointeur is access Integer;
  procedure Free is new Ada.Unchecked_Deallocation(Integer,T_Pointeur);
  Ptr1, Ptr2, Ptr3 : T_Pointeur;
begin
  Ptr1 := new Integer;
  Put("Quelle est la première valeur pointée ? ");
  get(Ptr1.all); skip_line;

  Ptr2 := new Integer;
  Put("Quelle est la seconde valeur pointée ? ");
  get(Ptr2.all); skip_line;

  Ptr3 := Ptr1;
  Ptr1 := Ptr2;
  Ptr2 := Ptr3; Free(Ptr3); --Attention à ne pas
  libérer Ptr1 ou Ptr2 !

  Put("Le premier pointeur pointe sur : ");
  Put(Ptr1.all); Free(Ptr1);
  Put(" et le second sur : ");
  Put(Ptr2.all); Free(Ptr2);
end Inverse_Pointeur;

```



Ce bête exercice d'inversion est important. Imaginez qu'u lieu d'inverser des Integer, il faille inverser des tableaux d'un bon million d'éléments (des données financières ou scientifiques par exemple), chaque élément comportant de très nombreuses informations. Cela peut être extrêmement long d'inverser chaque élément d'un tableau avec l'élément correspondant de l'autre. Avec des pointeurs, cela devient un jeu d'enfant puisqu'il suffit d'échanger deux adresses, ce qui ne représente que quelques octets.

Vous êtes parvenus à la fin de la première moitié de ce double-chapitre. Comme je vous l'avais dit en introduction, si des doutes subsistent, n'hésitez pas à relire ce chapitre ou à poser vos questions avant que les difficultés ne s'accumulent. La seconde moitié du chapitre sera en effet plus complexe. Elle vous apprendra à créer des pointeurs sur des variables, des constantes, des programmes... ou à créer des fonctions ou procédures manipulant des pointeurs. Le programme est encore chargé.

En résumé :

- Avant d'utiliser un pointeur, vous devez créer un type correspondant en indiquant le type des données qui seront pointées. Le terme correspondant est **ACCESS**.
- Un pointeur est une variable contenant l'adresse d'un emplacement mémoire. Les données utiles sont donc situées quelque part dans la mémoire de votre ordinateur et l'unique façon d'y accéder est d'utiliser son pointeur. Vous devez donc distinguer clairement le pointeur et l'élément pointé.
- Modifier la valeur d'un pointeur peut engendrer la perte de ces données, donc assurez-vous que vous ne perdrez pas l'adresse des données pointées : soit en utilisant un autre pointeur, soit en détruisant ces données à l'aide de `Unchecked_Deallocation()`.
- Les pointeurs apportent de sérieux avantages de par leur légèreté, mais exigent que vous preniez le temps de réfléchir au rôle de chacune de vos instructions et variables afin d'éviter un encocombrement inutile de la mémoire, la perte des données ou la modification imprudente des variables.

Les pointeurs II

Après un premier chapitre présentant la théorie sur les pointeurs (ou accesseurs) et l'allocation dynamique, nous allons dans cette seconde partie nous intéresser aux différentes possibilités offertes par le langage Ada : comment pointer sur une variable ou une constante déjà créée ? Comment créer des pointeurs constants ? Comment passer un pointeur en argument dans une fonction ou une procédure ? Comment créer puis manipuler des pointeurs sur des programmes ? C'est ce à quoi nous allons tenter de répondre dans ce chapitre avant d'effectuer quelques exercices d'application.

À noter que la troisième sous-partie (Pointeur sur un programme) est indiquée comme optionnelle. Elle est particulièrement compliquée et pourra être lue plus tard, quand vous aurez acquis une certaine aisance avec les pointeurs. Les notions qui y sont traitées sont importantes mais pas nécessaires pour la partie III. Il sera toutefois bon d'y revenir avant d'entamer la partie IV et notamment le chapitre sur les algorithmes de tri et la complexité.

Cas général

Pointeurs généralisés : pointer sur une variable

Pour l'instant, nous avons été obligés de pointer sur des valeurs définies par vos propres soins. Nous voudrions désormais pouvoir pointer sur une valeur générée par l'utilisateur ou par le programme, sans que le programmeur en ait préalablement connaissance. Une difficulté se pose : lorsque vous créez un « pointeur à », vous demandez à l'ordinateur de créer dans une zone non utilisée de la mémoire un emplacement contenant la valeur 7. Or pour pointer sur une variable, il ne faudra rien créer ! La variable est créée dès le démarrage de votre programme dans une zone de la mémoire qui a été préalablement réquisitionnée et dont la taille ne varie pas (c'est à cela que servent les déclarations). Il faut donc revoir notre type `T_Pointeur` pour spécifier qu'il peut pointer sur tout !

Code : Ada

```
Type T_Pointeur is access all integer;
```

L'instruction `ALL` permet d'indiquer que l'on peut pointer sur un emplacement mémoire créé en cours de programme (on parle de mémoire dynamique, comme pour l'allocation dynamique), mais aussi sur un emplacement mémoire déjà existant (on parle de mémoire statique) comme une variable.

La déclaration des pointeurs se fera normalement. En revanche, les variables que vous autoriserez à être pointées devront être clairement spécifiées pour éviter d'éventuels problèmes grâce à l'instruction `ALIASED`.

Code : Ada

```
Ptr : T_Pointeur;
N : aliased integer;
```



Et maintenant, comment je récupère l'adresse de ma variable ?

C'est simple, grâce à un attribut : `'ACCESS` !

Code : Ada

```
...
begin
    N := 374;           --N prend une valeur, ici 374
    Ptr := N'access;    --Ptr prend comme valeur l'adresse de N
```

Ainsi, `Ptr.all` correspondra à la variable `N` ! Retenez donc ces trois choses :

- le type `T_Pointeur` doit être généralisé avec `ALL` ;
- les variables doivent être spécifiées avec `ALIASED` ;
- l'adresse (ou plus exactement, la référence) peut être récupérée avec l'attribut `'ACCESS`.

Pointeur sur une constante et pointeur constant

Pointeur sur une constante

De la même manière, il est possible de créer des pointeurs pointant sur des constantes. Pour cela, notre type `T_Pointeur` doit être défini ainsi grâce à l'instruction `CONSTANT` :

Code : Ada

```
type T_Pointeur is access constant Integer;
```

Notre pointeur et notre constante seront donc déclarées ainsi :

Code : Ada

```
Ptr : T_Pointeur;
N : aliased constant integer := 35;
```

`Ptr` est déclaré de la même manière que d'habitude. Quant à notre constante `N`, elle doit être spécifiée comme pouvant être pointée grâce à l'instruction `ALIASED` ; elle doit être indiquée comme constante avec l'instruction `CONSTANT` et elle doit être indiquée comme de type `Integer` (bien sûr) valant 35 (par exemple). Puis, nous pourrons affecter l'adresse mémoire de `N` à notre pointeur `Ptr` :

Code : Ada

```
Ptr := N'access;
```



Ce qui est constant ici, ce n'est pas le pointeur mais la valeur pointée ! Il est donc possible par la suite de modifier la valeur de notre pointeur, par exemple en écrivant :

Code : Ada

```
Ptr := N'access;
Ptr := M'access;    --M doit bien-sûr être défini comme aliased
                    constant integer
```

En revanche il n'est pas possible de modifier la valeur pointée car c'est elle qui doit être constante. Vous ne pourrez donc pas écrire :

Code : Ada

```
X
  Ptr := N'Access;
  Ptr.all := 15;
  Ptr.all := N + 1;
```

Pointeur constant

Il est toutefois possible de créer des pointeurs constants. Redéfinissons de nouveau notre type `T_Pointeur` et nos objets et variables :

Code : Ada

```
type T_Pointeur is access all integer;
N : integer;
```

```
Ptr : constant T_Pointeur := N'access ;
```

Dans ce cas-ci, c'est le pointeur qui est constant : il pointera sur l'emplacement mémoire de la variable N et ne bougera plus. Maintenant, rien n'empêche N (et donc son emplacement mémoire) de prendre différentes valeurs (d'ailleurs, pour l'heure, Ptr pointe sur N mais N n'a aucune valeur). Exemple d'opérations possibles :

Code : Ada

```
N := 5 ;
Ptr.all := 9 ;
```

Exemple d'opérations interdites dès lors que notre pointeur est constant :

Code : Ada



```
Ptr := M'access ;
Ptr := new integer'(13) ;
```

Noubliez pas : ici c'est le pointeur qui est constant et donc l'adresse ne doit pas changer (par contre ce qui se trouve à cette adresse peut changer, ou pas).

Pointeur sur pointeur

L'idée semble saugrenue et pourtant elle fera l'objet de notre avant dernier chapitre : créer un pointeur sur un pointeur sur un entier ! Nous devrons donc créer deux types de pointeurs distincts : un type «pointeur sur integer» et un type «pointeur sur pointeur sur integer» !

Code : Ada

```
type T_Pointeur_Integer is access Integer ;
type T_Pointeur_Pointeur is access T_Pointeur_Integer ;

PtrI : T_Pointeur_Integer ;           --PtrI pour Pointeur-Integer
PtrP : T_Pointeur_Pointeur ;         --PtrP pour Pointeur-Pointeur
```

Cela devrait nous permettre de créer une sorte de liste de pointeurs qui se pointent les uns les autres (mais nous verrons cela plus en détail dans l'avant dernier chapitre) ! Attention, cela apporte quelques subtilités :

Code : Ada

```
PtrI := new Integer ;
PtrI.all := 78 ;

PtrP := new T_Pointeur_Integer ;
PtrP.all := PtrI ;
```

PtrP est un pointeur sur un pointeur, donc PtrP.all est lui aussi un pointeur, mais un pointeur sur un entier cette fois ! Si je souhaite modifier cet entier, deux solutions s'offrent à moi : soit utiliser PtrI.all, soit utiliser PtrP de la manière suivante :

Code : Ada

```
PtrP.all.all := 79 ;
```

Il serait même judicieux de reprendre notre code et de ne pas créer de pointeur PtrI. Tout devrait pouvoir être fait avec PtrP. Voyons cela en exemple :

Code : Ada

```
...
type T_Pointeur_Integer is access Integer ;
type T_Pointeur_Pointeur is access T_Pointeur_Integer ;
PtrP : T_Pointeur_Pointeur ;           --PtrP pour Pointeur-Pointeur
begin
  PtrP := new T_Pointeur_Integer ;
  PtrP.all := new Integer ;
  PtrP.all.all := 78 ;
...
```

Avec une seule variable PtrP, nous pouvons manipuler 3 emplacements mémoire :

- Emplacement mémoire du pointeur sur pointeur PtrP;
- Emplacement mémoire du pointeur sur integer PtrP.all;
- Emplacement mémoire de l'integer PtrP.all.all qui vaut ici 78 !

Et avec un pointeur sur pointeur sur pointeur sur pointeur... nous pourrions peut-être parvenir à créer, avec une seule variable, une liste d'informations liées les unes aux autres et de longueur infinie (ou tout du moins non contrainte, il suffirait d'utiliser l'instruction NEW à chaque fois que l'on souhaiterait allonger cette liste. Ces «listes» sont la principale application des pointeurs que nous verrons dans ce cours et fera l'objet d'un prochain chapitre ! Alors encore un peu de patience. 😊

Pointeur comme paramètre

Avant de passer aux pointeurs sur les programmes et aux exercices finaux, il nous reste un dernier point à traiter : comment transmettre un pointeur comme paramètre à une fonction ou une procédure ? Supposons que nous ayons deux procédures Lire() et Ecrire() et une fonction Calcul nécessitant un pointeur sur Integer en paramètre. Voyons tout d'abord comment déclarer nos procédures et fonctions sur des prototypes :

Code : Ada

```
procedure Lire(Ptr : T_Pointeur) ;
procedure Ecrire(Ptr : T_Pointeur) ;
function Calcul(Ptr : T_Pointeur) return Integer ;
```

Voici une autre façon de déclarer nos sous-programmes, permettant de s'affranchir du type T_Pointeur :

Code : Ada

```
procedure Lire(Ptr : access Integer) ;
procedure Ecrire(Ptr : access Integer) ;
function Calcul(Ptr : access Integer) return Integer ;
```

Avec cette seconde façon de procéder, il suffira de fournir en paramètre n'importe quel type de pointeur sur Integer, qu'il soit de type T_Pointeur ou autre. Par la suite, nous pourrons appeler nos sous-programmes de différentes façons :

Code : Ada

```
...
Ptr : T_Pointeur ;
N : aliased Integer := 10 ;
BEGIN
  Ptr := new Integer ;
  Lire(Ptr) ;
  Ecrire(N'access) ;
  N := Calcul(new integer'(25)) ;
...
```

Chacun de ces appels est correct. Mais il faut prendre une précaution : le pointeur que vous transmettez à vos

fonctions/procédures doivent absolument être initialisées et ne pas valoir **NULL** ! Faute de quoi, vous aurez le droit à un joli plantage de votre programme.

Seconde remarque d'importance : les pointeurs transmis en paramètres sont de mode **IN**. Autrement dit, la valeur du pointeur (adresse sur laquelle il pointe) ne peut pas être modifiée. Cependant, rien n'empêche de modifier la valeur pointée. **Ptr.all** n'est accessible qu'en lecture ; **Ptr.all** est accessible en lecture ET en écriture ! C'est donc comme si nous transmettions **Ptr.all** comme paramètre en mode **IN OUT**. Continuons la réflexion : pour les fonctions, il est normalement interdit de fournir des paramètres en mode **OUT** ou **IN OUT**, mais en fournit un pointeur comme paramètre, cela permet donc d'avoir, **dans une fonction**, un paramètre **Ptr.all** en mode **IN OUT** !

 Les pointeurs permettent ainsi de passer outre la rigueur du langage Ada. C'est à la fois un gain de souplesse mais également une grosse prise de risque. D'ailleurs, bien souvent, les projets Ada nécessitant une très haute sécurité s'interdisent l'emploi des pointeurs.

 La nouvelle norme Ada2012 permet désormais aux fonctions de manipuler des paramètres en mode **IN OUT**. Mais mon propos s'adressant au plus grand nombre, je tiens d'abord compte des normes les plus répandues Ada95 et Ada2005.

Pointeur sur un programme (optionnel)

Cette dernière partie théorique s'avère plus compliquée que les autres. Si vous pensez ne pas être encore suffisamment au point sur les pointeurs (joli jeu de mots !), je vous en déconseille la lecture pour l'instant et vous invite à vous exercer davantage. Si vous êtes suffisamment aguerris, alors nous allons passer aux choses très, TRÈS sérieuses ! 

Un exemple simple

Commençons par prendre un exemple simple (simplest même, mais rassurez-vous, l'exemple suivant sera bien plus aigu). Nous disposons de trois fonctions appelées **Double()**, **Triple()** et **Quadruple()** dont voici les spécifications :

Code : Ada

```
function Double(n : integer) return integer;
function Triple(n : integer) return integer;
function Quadruple(n : integer) return integer;
```

Peu nous importe de savoir comment elles ont été implémentées, ce que l'on sait c'est qu'elles nécessitent un paramètre de type **integer** et renvoient toutes un **integer** qui est, au choix, le double, le triple ou le quadruple du paramètre fourni. Seulement nous voudrions maintenant créer une procédure **Table()** qui affiche le double, le triple ou le quadruple de tous les nombres entre **a** et **b** (**a** et **b** étant deux nombres choisis par l'utilisateur, la fonction choisie l'étant elle aussi par l'utilisateur). Nous aurions besoin que les fonctions **double()**, **triple()** ou **quadruplet()** puissent être transmises à la procédure **Table()** comme de simples paramètres, ce qui éviterait que notre sous-procedure ait à se soucier du choix effectué par l'utilisateur. Nous voudrions ce genre de prototype :

Code : Ada

```
procedure Table(MaFonction : function;
                a,b : integer);
```

Malheureusement, cela n'est pas possible en tant que tel. Pour arriver à cela, nous allons devoir transmettre en paramètre, non pas une fonction, mais un pointeur sur une fonction. Donc première chose : comment déclarer un pointeur sur une fonction ? C'est avant tout des paramètres d'un certain type qui vont être modifiés afin d'obtenir un résultat d'un certain type. Ainsi, si un pointeur pointe sur une fonction à un paramètre, il ne pourra pas pointer sur une fonction à deux paramètres. Si un pointeur pointe sur une fonction renvoyant un **integer**, il ne pourra pas pointer sur une fonction renvoyant un **float**. Donc nous devons définir un type **T_Ptr_Fonction** qui prenne en compte tout cela : nombre et type de paramètres, type de résultat. Ce qui nous amène à la déclaration suivante :

Code : Ada

```
type T_Ptr_Fonction is access function(n : integer) return integer;
f : T_Ptr_Fonction;
-- est la notation mathématique usuelle pour fonction
-- mais n'oubliez pas qu'il s'agit uniquement d'un pointeur !
```

Définissons maintenant notre procédure **Table** :

Code : Ada

```
procedure Table(MaFonction : T_Ptr_Fonction;
                a,b : integer) is
begin
  for i in a..b loop
    put(i);
    put(" : ");
    put(MaFonction.all(i));
    new_line;
  end loop;
end Table;
```

Heureusement, le langage Ada nous permet de remplacer la ligne 7 par la ligne ci-dessous pour plus de commodité et de clarté :

Code : Ada

```
put(MaFonction(i));
```

Toutefois, n'oubliez pas que **MaFonction** n'est pas une fonction ! C'est un pointeur sur une fonction ! Ada nous fait grâce du **.all** pour les fonctions pointées à la condition que les paramètres soient écrits. Enfin, il ne reste plus qu'à appeler notre procédure **Table()** dans la procédure principale :

Code : Ada

```
--Choix de la fonction à traiter
put("Souhaitez-vous afficher les doubles(2), les triples(3) ou
les quadruples(4) ?");
get(choice); skip_line;
case choice is
  when 2 => f := double'access;
  when 3 => f := triple'access;
  when others => f := quadruple'access;
end case;

--Choix des bornes de l'intervalle
put("A partir de "); get(a); skip_line;
put("jusqu'à "); get(b); skip_line;

--Affichage de la table de a à b de la fonction f choisie
Table(f,a,b);
```

En procédant ainsi, l'ajout de fonctions **Quintuple()**, **Decuple()**, **Centuple()**... fonctionnant sur le même principe que les précédentes se fera très facilement sans modifier le code de la fonction **Table()**, mais uniquement la portion de code où l'utilisateur doit faire un choix entre les fonctions. Cela peut paraître anodin à notre niveau, mais ce petit gain de clarté et d'efficacité devient très important lors de projets plus complexes faisant intervenir plusieurs programmeurs. Celui rédigeant le programme **Table()** n'a pas besoin d'attendre que celui rédigeant les fonctions **Double()**, **Triplet()**... ait terminé, il peut dès lors et déjà coder sa procédure en utilisant une fonction / théorie.

 Il est également possible de créer des pointeurs sur des procédures. Bien évidemment, il est alors inutile de mentionner le résultat lors de la déclaration du type « Pointeur sur procédure ». Toutefois, prenez soin de vérifier que les modes des paramètres (**IN**, **OUT**, **IN OUT**) correspondent.

Un exemple de la vie courante

 Je ne vois toujours pas l'intérêt de créer des pointeurs sur des fonctions ?

Ah bon ? Pourtant vous utilisez en ce moment même ce genre de procédé. Alors même que vous lisez ces lignes, votre ordinateur doit gérer une liste de programmes divers et variés. Il vous suffit d'appuyer sur Alt+Ctrl+Suppr sous Windows puis de cliquer sur l'onglet Processus pour voir la liste des processus en cours. Sous Linux, ouvrez la console et tapez la commande ps (pour

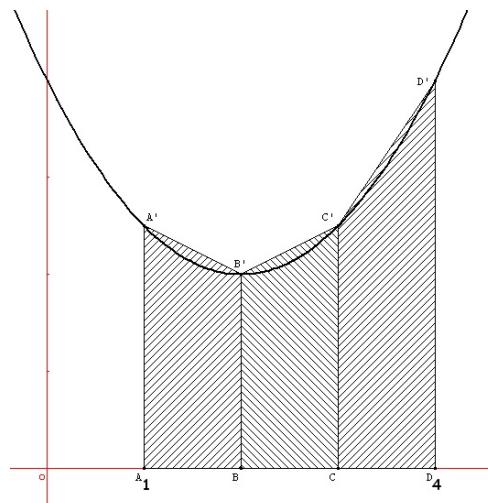
process status).

Comment votre système d'exploitation fait-il pour gérer plusieurs processus ? Entre Explorer, votre navigateur internet, votre antivirus, horloge... sans compter les programmes que vous pourrez encore ouvrir ou fermer, il faut bien que votre système d'exploitation (ce qui soit Windows, Linux, MacOs...) puisse gérer une liste changeante de programmes ! C'est, très schématiquement, ce que nous permettent les pointeurs : gérer des listes de longueurs indéfinies (voir le prochain chapitre) de pointeurs vers des programmes alternativement placés en mémoire vive ou traités par le processeur. Rassurez-vous, je ne compte pas vous faire développer un système d'exploitation dans votre prochain TP, je vous propose là simplement une illustration de notre travail.

Un exemple très... mathématique

Ce troisième et dernier exemple s'adresse à ceux qui ont des connaissances mathématiques un peu plus poussées (disons, niveau terminale scientifique). En effet, les fonctions sont un outil mathématique très puissant et très courant. Et deux opérations essentielles peuvent-être appliquées aux fonctions : la dérivation et l'intégration. Je souhaiterais traiter ici l'intégration. Ou plutôt, une approximation de l'intégrale d'une fonction par la méthode des trapèzes.

Pour rappel, et sans refaire un cours de Maths, l'intégrale entre a et b d'une fonction f correspond à l'aire comprise entre la courbe représentative de la fonction (entre a et b) et l'axe des abscisses. Pour approximer cette aire, plusieurs méthodes furent inventées, notamment la méthode des rectangles qui consiste à approcher l'aire de cette surface par une série de rectangles. La méthode des trapèzes consiste quant à elle à l'approximer par une série de trapèzes rectangles.



Graphique de la fonction f

Dans l'exemple précédent, on a cherché l'aire sous la courbe entre 1 et 4 . L'intervalle $[1 ; 4]$ a été partagé en 3 sous-intervalles de longueur 1 . La longueur de ces sous-intervalles est appelée le pas. Comment calculer l'aire du trapèze $AB'B'$? De la manière suivante : $\frac{AA' + BB'}{2} \times AB$. Et on remarque que $AA' = f(1)$ $BB' = f(1 + pas)$ $CC' = f(1 + 2 \times pas)$...

Nous allons donc définir quelques fonctions dont voici les prototypes :

Code : Ada

```
function carre(x : float) return float;
function cube(x : float) return float;
function inverse(x : float) return float;
```

Nous allons également définir un type `T_Pointeur` :

Code : Ada

```
type T_Pointeur is access function(x : float) return float;
```

Nous aurons besoin d'une fonction `Aire_Trapeze` et d'une fonction `Intégrale` dont voici les prototypes :

Code : Ada

```
--aire_trapeze() ne calcule l'aire que d'un seul trapèze
--f est la fonction étudiée
--x le "point de départ" du trapèze
--pas correspond au pas, du coup, x+pas est le "point d'arrivée" du
trapèze
function aire_trapeze(f : T_Pointeur; x : float; pas : float)
return float;

--integrale() effectue la somme des aires de tous les trapèzes
ainsi que le calcul du pas
--f est la fonction étudiée
--min et max sont les bornes
--nb_inter est le nombre de sous-intervalles et donc de trapèzes,
--ce qui correspond également à la précision de la mesure
function integrale(f : T_Pointeur;
min,max: float;
nb_inter : integer) return float is
```

Je vous laisse le soin de rédiger seuls ces deux fonctions. Cela constituera un excellent exercice sur les pointeurs mais aussi sur les intervalles et le pas. Il ne reste donc plus qu'à appeler notre fonction `integrale()` :

Code : Ada

```
--a,b sont des float saisis par l'utilisateur représentant les
bornes de l'intervalle d'intégration
--i est un integer saisi par l'utilisateur représentant le nombre
de trapèzes souhaités
integrale(carre'access,a,b,i);
integrale(cube'access,a,b,i);
integrale(inverse'access,a,b,i);
```

Bon allez ! Je suis bon joueur. Je vous transmets tout de même un code source possible. Attention, ce code est très rudimentaire, il ne teste pas si la borne inférieure est bel et bien inférieure à la borne supérieure, rien n'est vérifié quant à la continuité ou au signe de la fonction étudiée... bref, il est éminemment perfectible pour un usage mathématique régulier. Mais là, pas la peine d'insister, vous le ferez vraiment tous seuls. 😊

Secret (cliquez pour afficher)

Code : Ada

```
with ada.Text_IO, ada.Integer_Text_IO, ada.Float_Text_IO;
use ada.Text_IO, ada.Integer_Text_IO, ada.Float_Text_IO;

procedure integration is
-----  
-- Fonctions de x --  
-----
function carre(x : float) return float is
begin
```

```

    return x**2 ;
end carré ;

function inverse(x:float) return float is
begin
    return 1.0/x ;
end inverse ;

function cube(x:float) return float is
begin
    return x**3 ;
end cube ;

-----
-- Types --
-----

type T_Pointeur is access function (x : float) return float ;
-----  

-- Fonctions de f --
-----  

function aire_trapeze(f : T_Pointeur ; x : float ; pas : float)
return float is
begin
    return ((f.all(x)+f(x+pas))/2.0)*pas ;
end aire_trapeze ;

function integrale(f: T_pointeur ; min,max: float ; nb_inter :
integer) return float is
    res : float := 0.0 ;
    pas : float ;
begin
    pas := (max-min)/float(nb_inter) ;
    for i in 0..nb_inter-1 loop
        res := res + aire_trapeze(f,min+float(i)*pas,pas) ;
    end loop ;
    return res ;
end integrale ;
-----  

-- Procédure principale --
-----  

a,b : float ;
i : integer ;
begin
    put("Entrez quelles valeurs souhaitez-vous intégrer les
fonctions ?") ;
    get(a) ; get(b) ; skip_line ;
    put("Combien de trapèzes souhaitez-vous construire ?") ;
    get(i) ; skip_line ;
    put("L'intégrale de la fonction carré est : ") ;
    put(integrale(carré'access,a,b,i),Exp=>0) ; new_line ;
    put("L'intégrale de la fonction inverse est : ") ;
    put(integrale(inverse'access,a,b,i),Exp=>0) ; new_line ;
    put("L'intégrale de la fonction cube est : ") ;
    put(integrale(cube'access,a,b,i),Exp=>0) ; new_line ;
end integration ;

```

Exercices

Exercice 1

Énoncé

Voici un programme tout ce qu'il y a de plus simple :

Code : Ada

```

with ada.Text_IO, ada.Integer_Text_IO ;
use ada.Text_IO, ada.Integer_Text_IO ;

procedure programme is
n,m : integer ;
begin
    Put("Choisissez un nombre n :") ; get(n) ; skip_line ;
    Put("Choisissez un nombre m :") ; get(m) ; skip_line ;
    Put("La somme des nombres choisis est ") ;
    Put(n+m) ;
end programme ;

```

Vous devez modifier ce code de façon à changer le contenu des variables n et m, sans jamais modifier les variables elles-mêmes.
Ainsi, le résultat affiché sera faux.

Solution

Secret (cliquez pour afficher)

Code : Ada

```

with ada.Text_IO, ada.Integer_Text_IO ;
use ada.Text_IO, ada.Integer_Text_IO ;

procedure programme is
n,m : aliased integer ;
type T_Pointeur is access all integer ;
P : T_Pointeur ;
begin
    Put("Choisissez un nombre n :") ; get(n) ; skip_line ;
P := n'access ;
P.all := P.all + 1 ;
    Put("Choisissez un nombre m :") ; get(m) ; skip_line ;
P := m'access ;
P.all := P.all * 3 ;
    Put("La somme des nombres choisis est ") ;
    Put(n+m) ;
end programme ;

```

Exercice 2

Énoncé

Reprenez l'exercice précédent en remplaçant les deux variables n et m par un tableau T de deux entiers puis, en modifiant ses valeurs uniquement à l'aide d'un pointeur.

Solution

Secret (cliquez pour afficher)

Code : Ada

```

with ada.Text_IO, ada.Integer_Text_IO ;
use ada.Text_IO, ada.Integer_Text_IO ;

procedure programme is
    type T_Tableau is array(1..2) of integer ;
    type T_Pointeur is access all T_Tableau ;
    T : aliased T_Tableau ;
    P : T_Pointeur ;
begin
    Put("Choisissez un nombre n :") ; get(T(1)) ; skip_line ;
    Put("Choisissez un nombre m :") ; get(T(2)) ; skip_line ;
    P := T'access ;
    P.all(1) := P.all(1)*18 ;
    P.all(2) := P.all(2)/2 ;
    Put("La somme des nombres choisis est ") ;
    Put(T(1)+T(2)) ;

```

```
end programme ;
```

Exercice 3**Énoncé**

Reprenez le premier exercice, en remplaçant les deux variables n et m par un objet de type structuré comportant deux composantes entières puis, en modifiant ses composantes uniquement à l'aide d'un pointeur.

Solution**Secret** (cliquez pour afficher)

Code : Ada

```
with ada.Text_IO, ada.Integer_Text_IO;
use ada.Text_IO, ada.Integer_Text_IO;

procedure programme is
    type T_couple is
        record
            n,m : integer;
        end record;
    type T_Pointeur is access all T_couple;
    Couple : aliased T_couple;
    P : T_Pointeur;
begin
    Put("Choisissez un nombre n :"); get(couple.n); skip_line;
    Put("Choisissez un nombre m :"); get(couple.m); skip_line;
    P := Couple'access;
    P.all.n:= P.all.n*2;
    P.all.m:= P.all.m mod 3;
    Put("La somme des nombres choisis est ");
    Put(Couple.n + Couple.m);
end programme;
```

Il est également possible (mais je vous déconseille pour l'instant de le faire afin d'éviter toute confusion) d'écrire les lignes suivantes :

Code : Ada

 P.n:= P.n*2;
P.m:= P.m mod 3;

C'est un «raccourci» que permet le langage Ada. Attention toutefois à ne pas confondre le pointeur P et l'objet pointé !

Exercice 4**Énoncé**

Reprenez le premier exercice. La modification des deux variables n et m se fera cette fois à l'aide d'une procédure dont le(s) paramètres seront en mode **IN**.

Solution**Secret** (cliquez pour afficher)

Code : Ada

```
with ada.Text_IO, ada.Integer_Text_IO;
use ada.Text_IO, ada.Integer_Text_IO;

procedure programme is
    procedure modif(P : access integer) is
    begin
        P.all := P.all*5 + 8;
    end modif;
    n,m : aliased integer;
begin
    Put("Choisissez un nombre n :"); get(n); skip_line;
    modif(n'access);
    Put("Choisissez un nombre m :"); get(m); skip_line;
    modif(m'access);
    Put("La somme des nombres choisis est ");
    Put(n+m);
end programme;
```

Exercice 5 (Niveau Scientifique)**Énoncé**

Rédigez un programme calculant le coefficient directeur de la sécante à la courbe représentative d'une fonction *f* en deux points A et B. À partir de là, il sera possible de réaliser une seconde fonction qui donnera une valeur approchée de la dérivée en un point de la fonction *f*.

Par exemple, si $f(x) = x^2$ alors la fonction Secante(f,2.7) renverra $\frac{7^2 - 2^2}{7 - 2} = 9$. La fonction Derivee(f,5) renverra quant à elle $\frac{(5+h)^2 - 5^2}{h}$ où h devra être suffisamment petit afin d'améliorer la précision du calcul.

Solution**Secret** (cliquez pour afficher)

Code : Ada

```
with ada.Text_IO, ada.Float_Text_IO, ada.Integer_Text_IO;
use ada.Text_IO, ada.Float_Text_IO, ada.Integer_Text_IO;

procedure derivation is
    type T_Ptr_Fonction is access function(x : float) return float;
    function carre (x : float) return float is begin
        return x**2;
    end carre;
    function cube(x : float) return float is begin
        return x**3;
    end cube;
    function inverse(x : float) return float is begin
        return 1.0/x;
    end inverse;
    function secante(f : T_Ptr_Fonction;
                     a : float;
                     b : float) return float is
    begin
        return (f(b)-f(a))/(b-a);
    end secante;
    function derivee(f : T_Ptr_Fonction;
                     x : float;
                     precision : float := 0.000001) return float is
```

```

begin
    return secante(f,x,x+precision) ;
end derivee ;

choix : integer := 1 ;
f : T_Ptr_Fonction ;
x : float ;
begin
loop
    put("Quelle fonction souhaitez-vous dériver ? Inverse(1),
Carre(2) ou Cube(3) ") ;
    get(choix) ; skip_line ;
    case choix is
        when 1 => f := inverse'access ;
        when 2 => f := carre'access ;
        when 3 => f := cube'access ;
        when others => exit ;
    end case ;
    put("Pour quelle valeur de x souhaitez-vous connaître la
dérivée ?") ;
    get(x) ; skip_line ;
    put("La      derivee      vaut      environ") ;
    put(dérivee(x,x),Exp=>0,Aft=>3) ; put("      pour      x      =") ;
    put(x,Exp=>0,Aft=>2) ;
    new_line ; new_line ;
    end loop ;
end derivation ;

```

Il est possible de demander à l'utilisateur de régler la «précision» lui-même, mais j'ai retiré cette option car elle alourdissait l'utilisation du programme sans apporter de substantiels avantages.

Bien, si vous réalisez ces exercices c'est que vous devriez être venus à bout de ces deux chapitres éprouvants. Il est fort possible que vous soyiez encore assaillis de nombreuses questions. C'est pourquoi je vous conseille de pratiquer. Ce cours vous a proposé de nombreux exercices, il n'est peut-être pas inutile de reprendre des exercices déjà faits (affichage de tableau, de rectangles...) et de les retravailler à l'aide des pointeurs car cette notion est réellement compliquée. N'hésitez pas non plus à relire le cours à tête reposée.

Si d'aventure vous pensez maîtriser les pointeurs, alors un véritable test vous attend au chapitre 10. Nous mettrons les pointeurs en pratique en créant avec nos petites mains un nouveau type de donnée et les outils pour le manipuler : les types abstraits de donnée ! Là encore, ce sera un chapitre complexe, mêlant théorie et pratique. Mais avant cela nous allons aborder une notion difficile également : la récursivité.

En résumé :

- Si vous souhaitez que votre pointeur puisse également pointer sur des variables et donc sur de la mémoire statique, vous devrez le préciser lors de la déclaration du type en indiquant **IS ACCESS ALL**.
- De la même manière, si vous souhaitez qu'une variable puisse être pointée, il faudra l'indiquer par le terme **ALIASED**.
- Prenez soin de ne pas mélanger pointeur constant (où l'adresse pointée ne peut pas être modifiée, mais le contenu si) et pointeur sur une constante (où le pointeur peut être modifié à condition qu'il pointe toujours sur une constante).
- Le passage en paramètre d'un pointeur à une fonction permet d'avoir accès aux données pointées en lecture comme en écriture. Un paramètre en mode **ACCESS** est équivalent à un paramètre en mode **IN OUT**.
- Il est également possible de déclarer des pointeurs sur des fonctions ou des procédures. Mais vous devez faire attention au nombre, au type et au mode des paramètres (ou du résultat) lorsque vous déclarez le type pointeur.

Fonctions et procédures II : la récursivité

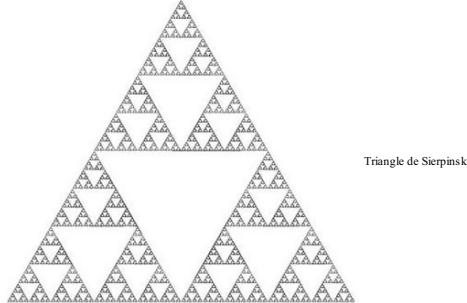
Le chapitre que vous vous apprêtez à lire est lui aussi compliqué, et pourtant vous n'aurez pas de nouvelle instruction ou de nouveau type composite à connaître.

 Mais alors quel intérêt de faire ce chapitre ? 

Nous allons aborder une notion d'algorithmique importante et qui nous servira pour le prochain chapitre : la **récursivité**. Ceux qui mènent (ou ont mené) des études scientifiques devraient être familiarisés avec un terme similaire : la « récurrence » ! Mais il n'y a pas d'erreur, la récursivité est une notion très proche du raisonnement par récurrence vu au lycée. Pour ceux qui n'ont pas mené ce genre d'étude, n'ayez aucune crainte, nous allons tout prendre à la base, comme toujours. 

Une première définition

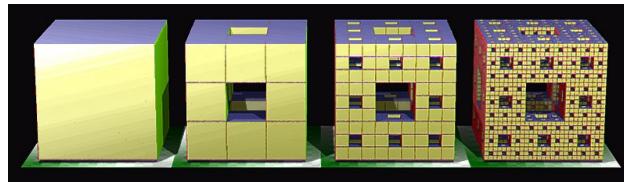
On dit qu'un programme est récursif si pour parvenir au résultat voulu il se réemploie lui-même. Cela peut s'illustrer par les images suivantes :



Pour réaliser le triangle de sierpinski (premier exemple), on relie les milieux des 3 côtés du grand triangle de manière à former trois triangles "noirs" et un triangle "blanc" à l'envers par rapport aux autres. Puis on reproduit cette opération sur les triangles noirs, et encore, et encore ... jusqu'au nombre désiré (Cette figure fait partie de la grande famille des **fractales** qui ont pour principe de répéter à l'infini sur elles-mêmes la même opération).



Idem pour le nautilus, on comprend que pour développer une coquille à 30 alvéoles, le nautilus doit avoir développé 29 alvéoles puis en créer seulement une supplémentaire de la même manière qu'il a créé les précédentes. Le nautilus répète ainsi sur lui-même à l'infini ou un grand nombre de fois, la même opération : «ajouter une alvéole». Un dernier exemple : l'éponge de Menger, une autre fractale inspirée du carré de Sierpinski. On a ici détaillé les 4 premières étapes :



Est-il besoin de réexpliquer cette fractale ?

Exemple d'algorithme récursif

 Bon, c'est gentil les illustrations, mais à part les coquillages et la géométrie, ça donne quoi en programmation ?

Imaginons que vous deviez créer un programme affichant un petit bonhomme devant monter N marches d'un escalier et que vous disposez pour cela d'une procédure appelée MonterUneMarche. Il y aura deux façons de procéder. Tout d'abord la manière itérative :

Code : Français

```
PROCÉDURE MonterALaMarche(n) :
  POUR i ALLANT de 1 à N
    | MonterUneMarche ;
  FIN DE BOUCLE
FIN DE PROCÉDURE
```

Ou alors on emploie une méthode récursive :

Code : Français

```
PROCÉDURE MonterALaMarche(n) :
  SI n > 0
    | MonterALaMarche(n-1) ;
    | MonterUneMarche ;
  FIN DE SI
FIN DE PROCÉDURE
```

La méthode récursive est un peu bizarre non ? Une explication s'impose. Si on veut monter 0 marche, c'est facile, il n'y a rien à faire. On a gagné ! En revanche, dans le cas contraire, il faut déjà avoir monté n-1 marches puis en gravir une autre. La procédure MonterMarche fait donc appel à elle-même pour une marche de moins. Prenons un exemple, si l'on souhaite monter 4 marches :

- 4 marches > 0 ! Donc on doit monter déjà 3 marches :
 - 3 marches > 0 ! Donc on doit monter déjà 2 marches :
 - 2 marches > 0 ! Donc on doit monter déjà 1 marche :
 - 1 marche > 0 ! Donc on doit monter déjà 0 marche :
 - 0 marche !!! c'est gagné !
 - Là on sait faire : le résultat renvoyé est que **marche = 0**.

- On incrémente marche : le résultat renvoyé est que `marche = 1`.
- On incrémente marche : le résultat renvoyé est que `marche = 2`.
- On incrémente marche : le résultat renvoyé est que `marche = 3`.
- On incrémente marche : le résultat renvoyé est que `marche = 4`.

Ce procédé est un peu plus compliqué à comprendre que la façon itérative (avec les boucles LOOP). Attention ! Comme pour les boucles itératives, il est possible d'engendrer une «boucle récursive infinie» ! Il est donc très important de réfléchir à la condition de terminaison. En récursivité, cette condition est en général le cas trivial, le cas le plus simple qu'il soit, la condition initiale : l'âge 0, la marche n°0, la lettre 'a' ou la case n°1 d'un tableau. Comme pour les boucles itératives, il est parfois utile de dérouler un exemple simple à la main pour être sûr qu'il n'y a pas d'erreur (souvent l'erreur se fait sur le cas initial).

Notre première fonction récursive

Enoncé

Nous allons créer une fonction `Produit(a,b)` qui effectuera le produit (la multiplication) de deux nombres entiers naturels (c'est-à-dire positifs) a et b . Facile me direz-vous. Sauf que j'ajoute une difficulté : il sera interdit d'utiliser le symbole $*$. Nous devrons donc implémenter cette fonction uniquement avec des additions et de manière récursive. Cet exemple peut paraître complètement idiot. Toutefois, il vous permettra de réaliser votre premier algorithme récursif et vous verrez que déjà, ce n'est pas de la tarte.

Si vous souhaitez vous lancer seuls, allez-y, je vous y invite. C'est d'ailleurs la meilleure façon de comprendre les difficultés. Simon, je vous donne quelques indications tout de suite, avant de vous révéler une solution possible.

Indications

Le cas trivial

Tout d'abord quel est le cas trivial, évident ? Autrement dit, quelle sera la condition de terminaison de notre boucle récursive ?

Le cas le plus simple est le cas `Produit(a,0)` ou `Produit(0,b)` puisqu'ils renvoient invariablement 0.



Je rappelle aux étourdis que $3 \times 0 = 0$ $0 \times 769 = 0$...

Il y a donc deux cas à traiter : si $a = 0$ ET/OU si $b = 0$.

Commutativité



Euh... c'est quoi encore ça ? Je comptais pas faire Bac +10 !!!

Pas d'inquiétude. La propriété de commutativité signifie juste que les nombres d'une multiplication peuvent être «commutés», c'est-à-dire échangés. Autrement dit $a \times b = b \times a$. En effet, que vous écrivez 2×3 ou 3×2 , cela reviendra au même : le résultat vaudra 6 ! C'est une propriété évidente qui ne bénéficiera pas la soustraction ou la division par exemple.



Si c'est si évident, pourquoi en parler ?

Et bien pour régler un problème simple : pour effectuer `Produit(2,3)`, c'est-à-dire 2×3 va-t-on faire $2 + 2 + 2$ (c'est-à-dire «3 fois» 2) ou $3 + 3$ (c'est-à-dire «2 fois» 3) ? Le deuxième choix semble plus rapide car il nécessite moins d'additions et donc moins de récursions (boucles récursives). Il serait donc judicieux de traiter ce cas avant de commencer. Selon que a sera inférieur ou pas à b , on effectuera `Produit(a,b)` ou `Produit(b,a)`.

Mise en œuvre

Supposons que a soit plus petit que b . Que doit faire `Produit(a,b)` (hormis le test de terminaison et d'inversion de a et b) ? Prenons un exemple : `Produit(3,5)`. Nous partons d'un résultat temporaire égal à 0 ($\text{res} = 0$).

- $3 > 0$. Donc notre résultat temporaire est augmenté de 5 ($\text{res} = 5$) et on fait le produit de 2 par 5 : `Produit(2,5)`.
 - $2 > 0$. Donc notre résultat temporaire est augmenté de 10 ($\text{res} = 10$) et on fait le produit de 1 par 5 : `Produit(1,5)`.
 - $1 > 0$. Donc notre résultat temporaire est augmenté de 15 ($\text{res} = 15$) et on fait le produit de 0 par 5 : `Produit(0,5)`.
 - 0 !!! Facile. On renvoie le résultat temporaire.

Se pose alors un souci : comment transmettre la valeur temporaire de `Res`, notre résultat ? La fonction `Produit` n'a que deux paramètres a et b ! L'idée, est de ne pas surcharger la fonction `Produit` inutilement avec des paramètres inutiles pour l'utilisateur final. Deux solutions s'offrent à vous : soit vous créez un paramètre `Res` initialisé à 0 (l'utilisateur n'aura ainsi pas besoin de le renseigner) ; soit vous créez une sous-fonction (`Pdt`, `Mult`, `Multiplication`, appelez-la comme vous voudrez) qui, elle, aura trois paramètres.



Nous allons voir juste après que la seconde façon a ma préférence.

Tests et temps processeur

Il est temps de faire un bilan : `Produit(a,b)` doit tester si a est plus petit que b ou égal car sinon il doit lancer `Produit(b,a)`. Puis il doit tester si $a = 0$, auquel cas il renvoie le résultat, sinon il doit lancer `Produit(a-1,b)`.

Nouveau problème, car en exécutant `Produit(a-1,b)`, notre programme va à nouveau tester si $a-1 < b$! Or si a était plus petit que b , $a-1$ ne risque pas de devenir plus grand. On effectue donc un test inutile. Ce pourrait être anodin, sauf qu'un test prend du temps au processeur et de la mémoire, et ce temps-processeur et cette mémoire seraient plus utiles pour autre chose. Si nous demandons le résultat de `Produit(7418,10965)`, notre programme devra effectuer 7418 tests inutiles ! Quel gâchis.

Deuxième et dernier bilan :

- Notre fonction `Produit(a,b)` se contentera de vérifier si a est plus petit ou égal à b . Et c'est tout. Elle passera ensuite le relais à une sous-fonction (que j'appellerai `Pdt`).
- Notre sous-fonction `Pdt` sera la vraie fonction récursive. Elle admettra trois paramètres : a , b et `res`. Le paramètre `a` devra être impérativement plus petit que b (c'est le travail de la fonction-mère) et `res` permettra de transmettre la valeur du résultat d'une récursion à l'autre.

Une solution possible

Le code ci-dessous ne constitue en aucun cas la seule et unique solution, mais seulement une solution possible :

Code : Ada

```
function Produit(a,b : natural) return natural is
  -----
  -- Inserer ici
  -- le code source de
  -- la fonction Pdt
  -----
begin
  if a <= b
    then return Pdt(a,b,0) ;
    else return Pdt(b,a,0) ;
  end if ;
end Produit ;
```

Pour plus de clarté, j'ai écrit une fonction `Produit` non récursive, mais faisant appel à une sous-fonction `Pdt` dont le code source n'est pas écrit. Cela permet à la fois d'effectuer la disjonction de cas et surtout de fournir une fonction `Produit` avec seulement deux paramètres. Rassurez-vous, je vous donne tout de même le code de la sous-fonction récursive :

Code : Ada

```
function Pdt(a,b,res : natural) return natural is
begin
  if a = 0
    then return res ;
    else return Pdt(a-1, b, res + b) ;
  end if ;
end Pdt ;
```



L'utilisateur final n'aura jamais accès directement à la fonction Pdt. Celle-ci étant écrite au sein même de la fonction Produit, sa durée de vie est restreinte à la fonction Produit. Impossible d'y accéder autrement.

Algorithmie de recherche par dichotomie

Voici désormais un nouveau programme à réaliser au cours duquel la récursivité va prendre tout son sens. Le principe est d'implémenter un programme «Dichotomie()» qui recherche un nombre entier dans un tableau préalablement classé par ordre croissant (et sans doublons).

Principe

Par exemple, nous souhaiterions savoir où se trouve le nombre 9 dans le tableau suivant :

1	2	3	4	5	6	7	8	9
0	2	3	5	7	9	15	20	23

Une méthode de bournin consiste à passer tous les nombres du tableau en revue, de gauche à droite. Dans cet exemple, il faudra donc tester les 6 premiers nombres pour savoir que le 6ème nombre est un 9 ! Comme je vous l'ai dit, 6 tests, c'est une méthode de bournin. Je vous propose de le faire en 3 étapes seulement avec la méthode par dichotomie (prononcez «dihotomie»).



Ça veut dire quoi dichotomie ?

Ce mot vient du grec ancien et signifie «couper en deux». On retrouve le suffixe «tomie» présent dans les mots appendicectomie (opération consistant à couper l'appendice), mammectomie (opération consistant à couper un sein pour lutter contre le cancer du même nom), lobotomie (opération consistant à couper et retirer une partie du cerveau)... Bref, on va couper notre tableau en deux parties (pas nécessairement égales). Pour mieux comprendre, nous allons dérouler un algorithme par dichotomie avec l'exemple ci-dessus.

Tout d'abord, on regarde le nombre du «milieu», c'est-à-dire le numéro 5 (opération : $\frac{1+9}{2} = 5$)

1	2	3	4	6	7	8	9	
0	2	3	5	7	9	15	20	23

Comme 7 est plus petit que 9, on recommence mais seulement avec la partie supérieure du tableau car comme le tableau est ordonné, il ne peut y avoir de 9 avant !

6	7	8	9
9	15	20	23

On prend à nouveau le nombre du «milieu». L'opération $\frac{6+9}{2}$ donne 7,5 ! Donc nous allons considérer que le nombre du «milieu» est le n°7 (je sais, ce n'est pas mathématiquement rigoureux, mais c'est bien pour cela que j'utilise des guillemets depuis le début).

6	7	8	9
9	15	20	23

Le nombre obtenu (15) est plus grand que 9, on recommence donc avec la partie inférieure du tableau.

6
9

Le nombre du «milieu» est le n°6 (logique) et il vaut 9. C'est gagné ! Notre résultat est que «le 9 est à la 6ème case».

9
9

Avec cet algorithme, je n'ai testé que la 5ème, la 7ème et la 6ème valeur du tableau, soit seulement 3 tests au lieu de 6 pour la méthode «bournin» (on parle de force brute). Bref, cet écart peut très vite s'accroître notamment pour rechercher de grandes valeurs dans un grand tableau. Nous verrons dans le chapitre sur la complexité des algorithmes que le premier algorithme a une complexité en $O(n)$ tandis que le second a une complexité en $O(\log(n))$. Oui, je sais, pour vous c'est du Chinois. Nous expliquerons cela plus tard, retenez seulement que cela signifie que pour un petit tableau, l'intérêt de la dichotomie est discutable ; en revanche pour un grand ou très grand tableau, il n'y a pas photo sur la plus grande efficacité de la dichotomie.

Mise en œuvre

Nous allons désormais implémenter notre programme «Dichotomie()». Toutefois, vous aurez remarqué qu'il utilise des tableaux de tailles différentes. Nous avons donc un souci car nous ne savons déclarer que des tableaux de taille fixe. Deux solutions sont possibles :

- Regarder le chapitre de la partie 4 sur la généricité pour pouvoir faire des tableaux dont on ne connaît pas préalablement la taille. Le problème, c'est que ton risque de vite te mélanger les pinceaux, d'autant plus que ce chapitre est dans une partie encore plus dure.
- Faire en sorte que notre programme manipule un tableau T de taille fixe mais aussi deux variables Min et Max qui indiqueront la « plage » du tableau qui nous intéresse (*cherche dans le tableau T entre l'indice Min et l'indice Max*). Nous retiendrons pour instant, vous vous en doutez, cette seconde solution. Toutefois, il sera judicieux de reprendre ce programme après avoir vu la généricité.

Autre question : que donne la division 15/2 en Ada ? Réponse :

Code : Console

Reponse : 7_

Puisque nous divisons deux integer, Ada nous renvoie également un integer. Or vous savez bien que cette division ne «tombe pas juste» et qu'elle devrait donner 7,5 ! Mais Ada doit faire un choix : soit la valeur approchée à l'unité près par excès (8) soit la valeur approchée à l'unité près par défaut (7). Le choix a été fait de renvoyer la valeur par défaut car elle correspond à la partie entière du nombre décimal. Toutefois, si vous voulez savoir durant la programmation de cet algorithme, si un nombre est pair ou impair, je vous rappelle qu'il existe l'opération MOD !

Exemple : 15 mod 2 = 1 (Donc 15 est impair) ; 18 mod 2 = 0 (Donc 18 est pair).

Dernier problème à régler : que faire si l'il n'y a pas de 9 ? Renvoyer une valeur particulière ? Ce n'est pas la meilleure idée. L'utilisateur final n'est pas sensé le savoir. Renvoyer l'indice la valeur la plus proche ? Pourquoi pas mais si l'utilisateur ne connaît pas cette nuance, cela peut poser problème. Le mieux serait de renvoyer deux résultats : un booléen Existé indiquant si la valeur existe ou non dans le tableau et une variable Indice indiquant la valeur trouvée (seulement si Existé vaut true).

Dès lors, plusieurs solutions s'offrent à vous : vous pouvez créer un type structuré (avec RECORD) pour renvoyer deux résultats en même temps. Ou vous pouvez créer une procédure avec deux paramètres en mode OUT ou encore utiliser un pointeur sur booléen comme paramètre à votre fonction pour savoir savoir si le résultat existe. A vous de choisir.

Solution

Voici une solution possible pour le programme «Dichotomie()» :

Code : Ada

```
Procedure Dichotomie is
    type T_Tableau is array(1..50) of integer;           --
    -- à vous de choisir la taille du tableau
    type T_Ptr_Boolean is access boolean;
    -----
    -- Insérer ici le code source --
    -- de la fonction récursive --
    -----
T : T_Tableau;
```

```

Existe : T_Ptr_Bool ;
Indice : integer ;

begin
  Init(T) ; -- à vous d'initialiser ce tableau
  Existe := new boolean ; -- création et initialisation du pointeur
  Existe.all := false ;
  Indice := Dicho(T, 13, Existe, T.first, T.last) ; -- on cherche le nombre 13 dans le tableau T
  if Existe.all then put("Le nombre 13 est à l'indice numero") ;
    put(Indice) ;
  else put("Le nombre 13 n'est pas présent dans le tableau") ;
  end if ;
end Dicho;

```

Dans le programme ci-dessus, on recherche le nombre 13 dans un tableau de 50 cases. La procédure d'initialisation n'est pas écrite, je vous laisse le faire (pensez que le tableau doit être ordonné). Voici maintenant le code de la fonction Dicho() :

Code : Ada

```

function Dicho(T : T_Tableau ;
  N : integer ;
  Exist : T_Ptr_Bool ;
  Min, Max : integer) return integer is
  Milieu : integer ;
begin
  Milieu := (Min+Max) / 2 ;
  if T(Milieu) = N then -- Cas où on trouve le nombre N cherché
    Existe.all := true ;
    return Milieu ;
  elsif Min = Max then -- Cas où l'on n'a pas trouvé N et où on ne peut plus continuer
    Existe.all := false ;
    On indique que N n'existe pas et on renvoie 0.
    return 0 ;
  elsif T(Milieu) > N then -- Cas où on peut continuer avec un sous-tableau
    then return Dicho(T,N,Exist,Min,Milieu-1) ;
    else return Dicho(T,N,Exist,Milieu+1,Max) ;
  end if ;
end Dicho ;

```

Nous avons cette fois réalisé un programme où l'utilisation de la récursivité est vraiment pertinente. À quoi reconnaît-on qu'un programme pourra être traité par une récursivité ? Tout d'abord il faut avoir besoin de répéter certaines actions. La question est plutôt : pourquoi choisir un algorithme récursif plutôt qu'un itératif ? La réponse est en grande partie dans la définition même de la récursivité. Il faut se trouver face à un problème qui ne se résolve qu'en réappelant les mêmes procédés sur un même objet. Les algorithmes récursifs nécessitent généralement de disjoindre deux cas : le cas simple, trivial et le cas complexe. Ce dernier peut être lui-même subdivisé en plusieurs autres cas (a<b et a>b ; n pair et n impair...).

Quelques exercices

Exercice 1

Énoncé

Rédigez une fonction récursive Factorielle() qui calcule... la factorielle du nombre donné en argument. Pour exemple, la factorielle de 7 est donnée ainsi : $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$. Ainsi, votre fonction Factorielle(7) renverra 5040.

Solution

Secret (cliquez pour afficher)

Code : Ada

```

function factorielle(n : integer) return integer is
begin
  if n>0 then return n*factorielle(n-1) ;
  else return 1 ;
  end if ;
end factorielle ;

```

⚠ À partir de $17!$, des problèmes commencent à apparaître : des résultats sont négatifs ce qui est impossible puisqu'on ne multiplie que des nombres positifs. De plus, les résultats ne « grossissent » plus.

Alors pourquoi ces résultats ? C'est ce que l'on appelle l'*overflow*. Je vous ai souvent mis en garde : l'infini en informatique n'existe pas, les machines ont beau être ultra-puissantes, elles ont toujours une limite physique. Par exemple, les integer sont codés par le langage Ada sur 32 bits, c'est-à-dire que l'ordinateur ne peut retenir que des nombres formés de maximum 32 chiffres (ces chiffres n'étant que des 0 ou des 1, c'est ce que l'on appelle le **code binaire**, seul langage compréhensible par un ordinateur). Pour être plus précis même, le 1er chiffre (on dit le premier bit) correspond au signe : 0 pour positif et 1 pour négatif. Il ne reste donc plus que 31 bits pour coder le nombre. Le plus grand integer enregistrable est donc $2^{31} - 1 = 2147483647$. Si jamais nous enregistrons un nombre plus grand encore, il nécessiterait plus de bits que l'ordinateur ne peut en fournir, c'est ce que l'on appelle un dépassement de capacité ou overflow. Du coup, le seul bit supplémentaire possible est celui du signe + ou -, ce qui explique les erreurs obtenues à partir de 17.

Exercice 2

Énoncé

Rédigez une fonction récursive Puissance(a,n) qui calcule la puissance n du nombre a (c'est-à-dire a^n), mais seulement en utilisant des multiplications.

Solution

Secret (cliquez pour afficher)

Code : Ada

```

function puissance(a,n : integer) return integer is
begin
  if n > 0 then return a*puissance(a,n-1) ;
  else return 1 ;
  end if ;
end puissance ;

```

⚠ Si vous cherchez à calculer 17^{11} , vous obtiendrez là aussi un résultat négatif du à un overflow.

Exercice 3

Énoncé

Rédigez des fonctions récursives Affichage() et Minimum(). Chacune devra parcourir un tableau, soit pour l'afficher soit pour indiquer l'indice du plus petit élément du tableau.

Solution

Secret (cliquez pour afficher)

Code : Ada

```

type T_Tableau is array(1..10) of integer ;
procedure Affichage(T : T_Tableau) is
begin
  put(T(dbt),1); put(" ; ");
  if dbt < T'last
    then Afg(T,dbt+1);
  end if;
end Afg;
begin
  Afg(T,T'first);
end Affichage;

function Minimum(T:T_Tableau) return integer is
  function minimum(T:T_Tableau; rang, res, min : integer) return
integer is
    --rang est l'indice que l'on va tester
    --res est le résultat temporaire, l'emplacement du
minimum trouvé jusque là
    --min est le minimum trouvé jusque là
    min2 : integer;
    res2 : integer;
begin
  if T(rang) < min
    then min2 := T(rang);
    res2 := rang;
  else min2 := min;
    res2 := res;
  end if;
  if rang = T'last
    then return res2;
    else return minimum(T,rang+1,res2,min2);
  end if;
end minimum;
begin
  return minimum(T,T'first+1,T'first,T(T'first));
end Minimum;

```



Il existe deux fonctions minimum, l'une récursive et l'autre non ! Mais cela ne pose pas de problème car le compilateur constate que le nombre de paramètres est différent. Il est donc facile de les distinguer. De plus, dans votre procédure principale, seule la fonction minimum non récursive sera accessible.

Exercice 4**Énoncé**

Rédigez une fonction récursive Effectif(T,x) qui parcourt un tableau T à la recherche du nombre d'apparition de l'élément x. Ce nombre d'apparition est appelé effectif. En le divisant par le nombre total d'élément dans le tableau, on obtient la fréquence (exprimée en pourcentages si vous la multipliez par 100). Cela vous permettra ainsi de rédiger une seconde fonction appelée Fréquence(T,x). Le tableau T pourra contenir ce que vous souhaitez, il peut même s'agir d'un string.

Solution

Secret (cliquez pour afficher)

Code : Ada

```

function effectif(T: T_Tableau ; x : integer) return integer is
  function effectif(T:T_Tableau ; x : integer; rang, eff :
integer) return integer is
    eff2 : integer;
begin
  if T(rang) = x
    then eff2 := eff + 1;
    else eff2 := eff;
  end if;
  if rang = T'last
    then return eff2;
    else return effectif(T,x,rang+1,eff2);
  end if;
end effectif;
begin
  return effectif(T,x,T'first,0);
end effectif;

function frequence(T : T_Tableau ; x : integer) return float is
begin
  return float(effectif(T,x)/float(T'length)*100.0;
end frequence;

```



Là encore, je crée deux fonctions effectif() de manière à ce que l'emploi de la première (non récursive) soit facilité (peu de paramètres). La seconde fonction effectif(), récursive et «coincée» dans la première, nécessite plus de paramètres qui n'ont pas à être connus par l'utilisateur final de ce code.



Nous venons de créer plusieurs fonctions ou procédures liées aux tableaux. Peut-être pourriez-vous les ajouter à votre package P_Integer_Array.

Bien, nous avons désormais terminé ce chapitre sur la récursivité. Cette notion va nous être utile au prochain chapitre puisque nous allons aborder un nouveau type composite : les types abstraits de données. Ce chapitre fera appel aux pointeurs, aux types structurés et à la récursivité. Ce sera le dernier chapitre théorique de cette partie et probablement le plus compliqué, donc si vous avez encore des difficultés avec l'une de ces trois notions, je vous invite à relire les chapitres précédents et à vous entraîner car les listes sont des objets un peu plus compliqués à manipuler.

J'espère que ce chapitre vous aura permis d'avoir une idée assez précise de ce qu'est la récursivité et de l'intérêt qu'elle peut représenter en algorithmique. Pour disposer d'un second point de vue, je vous conseille également la lecture du tutoriel de bluestorm disponible sur le site du zéro en cliquant [ici](#). Toutefois, s'il est plus complet concernant la notion de récursivité, ce tutoriel n'est pas rédigé en Ada, vous devrez donc vous contenter de ses explications.

En résumé :

- Les boucles récursives permettent, comme les boucles itératives, de répéter une tâche. Il suffit pour cela qu'une procédure ou une fonction s'appelle elle-même.
- Un programme récursif doit envisager plusieurs cas et notamment le cas trivial qui mettra fin à la boucle récursive. Une fois ce cas évident établi, vous devez réfléchir à l'opération à effectuer pour passer à l'étape supérieure.
- Les algorithmes récursifs nécessitent plus d'entraînement et de réflexion que les algorithmes itératifs. Nous verrons au prochain chapitre que certaines structures s'adaptent mieux à ce type de raisonnement. C'est pourquoi vous devez malgré tout vous entraîner à la récursivité, malgré les difficultés qu'elle peut présenter.

Les Types Abstraits de Données : listes, files, piles

...

Dans ce chapitre, nous allons aborder un nouveau type composite : les Types Abstraits de Données, aussi appelés TAD. De quoi s'agit-il ? Eh bien pour le comprendre, il faut partir du problème qui amène à leur création. Nous avons vu au début de la partie III le type **ARRAY**. Les tableaux sont très pratiques mais ont un énorme inconvénient : ils sont contraints. Leur taille est fixée au début du programme et ne peut plus bouger. Imaginons que nous réalisions un jeu d'aventure. Notre personnage découvre des trésors au fil de sa quête, qu'il ajoute à son coffre à trésor. Ce serait idiot de créer un objet coffre de type Tableau. S'il est prévu pour contenir 100 items et que l'on en découvre 101, il sera impossible d'enregistrer notre 101ème item dans le coffre ! Il serait judicieux que le coffre puisse contenir autant de trésors qu'on le souhaite.

 Mais il existe bien des `unbounded_string` ??? Alors il y a peut-être des `unbounded_array` ?

Eh bien non, pas tout à fait. Nous allons donc devoir en créer nous mêmes, avec nos petits doigts : des «listes» d'éléments possiblement infinies. C'est ce que l'on appelle les TAD. Nous ferons un petit tour d'horizon théorique de ce que sont les Types Abstraits de Données. Puis nous créerons dans les parties suivantes quelques types abstraits avant de voir ce que le langage Ada nous propose. Nous créerons également les outils minimaux nécessaires à leur manipulation avant de voir un exemple d'utilisation.

Qu'est-ce qu'un Type Abstrait de Données ?

Un premier cas

Comme je viens de vous le dire, les Types Abstraits de Données peuvent s'imaginer comme des «listes» d'éléments. Les tableaux constituent un type possible de TAD. Mais ce qui nous intéresserait davantage, ce serait des «listes infinies». Commençons par faire un point sur le vocabulaire employé. Lorsque je parle de «liste», je commets un abus de langage. En effet, il existe plusieurs types abstraits, et les listes (je devrais parler plus exactement de «listes chaînées» par distinction avec les tableaux) ne constituent qu'un type abstrait parmi tant d'autres. Mais pour mieux comprendre ce que peut être un TAD, nous allons pour l'instant nous accommoder de cet abus (je mettrai donc le mot liste entre guillemets). Voici une représentation possible d'un Type Abstrait de Donnée :

- «**Liste** vide : un emplacement mémoire a été réquisitionné pour la liste, mais il ne contient rien.



Liste vide

- «**Liste** contenant un élément : le premier emplacement mémoire contient un nombre (ici 125) et pointe sur un second emplacement mémoire, encore vide pour l'instant.



Liste contenant un élément

- «**Liste** contenant 3 éléments : trois emplacements mémoire ont été réquisitionnés pour contenir des nombres qui se sont ajoutés les uns à la suite des autres. Chaque emplacement pointe ainsi sur le suivant. Le troisième pointe quant à lui sur un quatrième emplacement mémoire vide.



Liste contenant 3 éléments

Autres cas

L'exemple ci-dessus est particulier, à plusieurs égards. Il n'existe pas qu'un seul type de «liste» (et je ne compte pas les traiter toutes). La «liste» vue précédemment dispose de certaines propriétés qu'il nous est possible de modifier pour en élaborer d'autres types.

Contraintes de longueur

La «liste» vue ci-dessus n'a pas de contrainte de longueur. On peut y ajouter autant d'éléments que la mémoire de l'ordinateur peut en contenir mais il est impossible d'obtenir le 25ème élément directement, il faut pour cela le parcourir à partir du début. Les tableaux sont quant à eux des listes contraintes et indexées de manière à accéder directement à tout élément, mais la taille est donc définie dès le départ et ne peut plus être modifiée par la suite. Mais vous aurez compris que dans ce chapitre, les tableaux et leurs contraintes de longueur ne nous intéressent guère.

Chainage

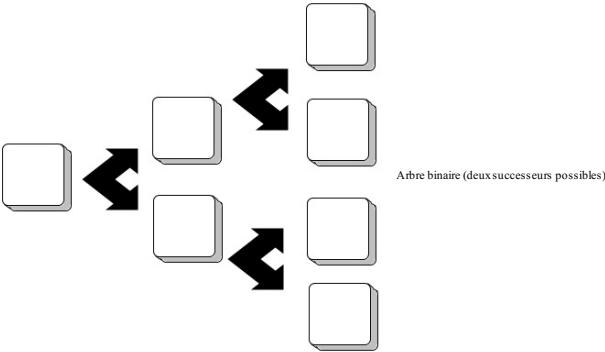
Chaque élément pointe sur son successeur. On dit que la «liste» est **simplement chaînée**. Une variante serait que chaque élément pointe également sur son prédécesseur. On dit alors que la «liste» est **doubllement chaînée**.



Liste simplement chaînée

Linéarité

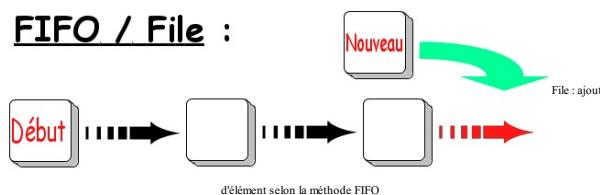
Chaque élément ne pointe que sur un seul autre élément. Il est possible de diversifier cela en permettant à chaque élément de pointer sur plusieurs autres (que ce soit un nombre fixe ou illimité d'éléments). On parlera alors d'**arbre** et non plus de «liste».



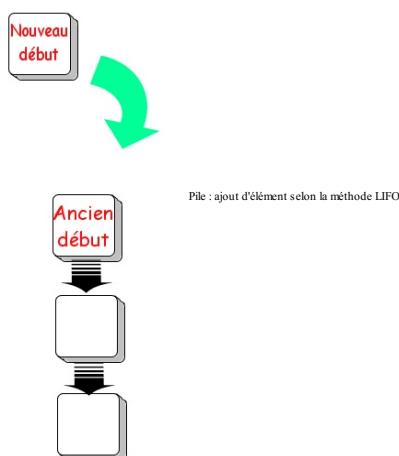
Arbre binaire (deux successeurs possibles)

Méthode d'ajout

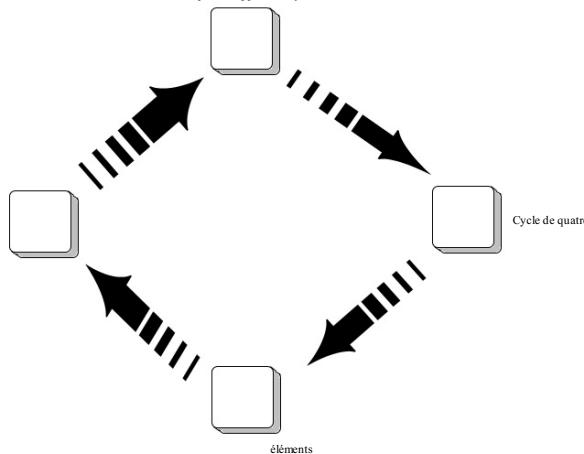
Chaque élément ajouté l'a été en fin de «liste». Cette méthode est dite FIFO pour First In/First Out, c'est-à-dire «premier arrivé, premier servi». Le second élément ajouté sera donc en deuxième position, le troisième élément en troisième position... notre «liste» est ce que l'on appelle une **file** (en Français dans le texte). C'est ce qui se passe avec les imprimantes reliées à plusieurs PC : le premier à appuyer sur "Impression" aura la priorité sur tous les autres. Logique, me direz-vous.

FIFO / File :

Mais ce n'est pas obligatoirement le cas. La méthode «inverse» est dite LIFO pour «Last In/First Out», c'est-à-dire «dernier arrivé, premier servi». Le premier élément ajouté sera évidemment le premier de la «liste». Lorsqu'on ajoute un second élément, c'est lui qui devient le premier de la «liste» et l'ex-premier devient le second. Si on ajoute un troisième élément, il sera le premier de la «liste», l'ex-premier devient le second, l'ex-second devient le troisième... on dirait alors que notre liste est une **pile**. On peut se représenter la situation comme une pile d'assiettes, la dernière que vous ajoutez devant la première de la **pile**. L'historique de navigation de votre navigateur internet agit selon une pile : lorsque vous cliquez sur précédent, la première page qui s'affichera sera la dernière à laquelle vous avez accédé. La «liste» des modifications apportées à un document est également gérée telle une pile : cliquez sur Annuler et vous annulerez la dernière modification.

LIFO / Pile :**Cyclicité**

Le dernier élément de notre liste ne pointe sur rien du tout ! Or nous pourrions faire en sorte qu'il pointe sur le premier élément. Il n'y a alors ni début, ni fin : notre liste est ce que l'on appelle un **cycle**.



Il est bien sûr possible de combiner différentes propriétés : arbres partiellement cycliques, cycles doublement chaînés... pour élaborer de nouvelles structures.

Primitives

Mais un type abstrait de données se définit également par ses **primitives**. De quoi s'agit-il ? Les primitives sont simplement les fonctions essentielles s'appliquant à notre structure. Ainsi, pour une file, la fonction Ajouter_a_la_fin() constitue une primitive. Pour une pile, ce pourrait être une fonction Ajouter_sur_la_pile().

En revanche, une fonction Ranger_a_l_envers(p : pile) ou Mettre_le_chantier(f : file) ou Reduire_le_nombre_de_branches(arbre) ne sont pas des primitives : ce ne sont pas des fonctions essentielles au fonctionnement et à l'utilisation de ces structures (même si ces fonctions ont tout à fait le droit d'exister).

Donc créer un TAD tel une pile, une file... ce n'est pas seulement créer un nouveau type d'objet, c'est également fournir les primitives qui permettront au programmeur final de le créer, de le détruire, de le manipuler... sans avoir besoin de connaître sa structure. C'est cet ensemble TAD-Primitives qui constitue ce que l'on appelle une **Structure de données**.

Mais alors, qu'est-ce que c'est exactement qu'une liste (sans les guillemets) ? Tu nous a même parlé de liste chaînée : c'est quoi ?

Une liste chaînée est une structure de données (c'est à dire un type abstrait de données fourni avec des primitives) linéaire et non contrainte. Elle peut être cyclique ou non, simplement ou doublement chaînée. L'accès aux données doit pouvoir se faire librement (ni en FIFO, ni en LIFO) ; il doit être possible d'ajouter/modifier/retirer n'importe quel élément sans avoir préalablement retiré ceux qui le précédaient ou le suivaient. Toutefois, il faudra très souvent parcourir la liste chaînée depuis le début avant de pouvoir effectuer un ajout, une modification ou une suppression.

Pour mieux comprendre, je vous propose de créer quelques types abstraits de données (que je nommerai désormais TAD, je sais je suis un peu fanéant). Je vous propose de construire successivement une pile puis une file avant de nous pencher sur les listes chaînées. Cela ne conviendra sûrement pas tout le champ des TAD (ce cours n'y suffirait pas) mais devrait vous en donner une bonne idée et vous livrer des outils parmi les plus utiles à tout bon programmeur.

Les piles

Nous allons tout d'abord créer un type `T_Pile` ainsi que les primitives associées. Pour faciliter la réutilisation de ce type composite, nous écrirons tout cela dans un package que nous appellerons `P_Piles`. Vous remarquerez les quelques subtilités d'écriture : `P_Piles` pour le package ; `T_Pile` pour le type ; `Pile` pour la variable. Ce package comportera donc le type `T_Pile` et les primitives. Nous ajouterons ensuite quelques procédures et fonctions utiles (mais qui ne sont pas des primitives).

Création du type T_Pile

Tout d'abord, rappelons succinctement ce que l'on entend par «pile». Une pile est un TAD linéaire, non cyclique, simplement chaîné et non contraint. Tout ajout/suppression se fait par la méthode LIFO : Last In, First Out ; c'est à dire que le programmeur final ne peut accéder qu'à l'élément placé sur le dessus de la pile. Alors, comment créer une liste infinie de valeurs liées les unes aux autres ? Nous avions dès lors déjà approché cette idée lors du précédent chapitre sur les pointeurs. Il suffirait d'un pointeur pointant sur un pointeur pointant sur un pointeur... Mais, premier souci, où et comment enregistrer-t-on nos données s'il n'y a que des pointeurs ? Une idée ? Avec un type structuré bien sûr ! Schématiquement, nous aurions ceci :

Code : Français

```
DECLARATION DE MonType :
  Valeur : integer, float ou que sais-je ;
  Pointeur_vers_le_prochain : pointeur vers MonType ;
FIN DE DECLARATION
```

Nous pourrions également ajouter une troisième composante (appelée Index ou Indice) pour numérotter nos valeurs. Nous devrons également déclarer notre type de pointeur ainsi :

Code : Français

```
TYPE T_Pointeur IS ACCESS MonType ;
```

Sauf que, deuxième souci, si nous la déclarons après MonType, le compilateur va se demander à quoi correspond la composante Pointeur_vers_le_prochain. Et si nous déclarons T_Pointeur avant MonType, le compilateur ne comprendra pas vers quoi il est sensé pointé. ☺

Comment résoudre ce dilemme ? C'est le problème de l'œuf et de la poule : qui doit apparaître en premier ? Réfléchissez. Là encore, vous connaissez la solution. Nous l'avons employée pour les packages : il suffit d'employer les spécifications ! Il suffit que nous déclarions MonType avec une spécification (pour rassurer le compilateur ☺), puis le type T_Pointeur et enfin que nous déclarions vraiment MonType. Compris ? Alors voilà le code correspondant :

Code : Ada

```
TYPE T_Cellule;           --T_Cellule correspond à un
                           élément de la pile
TYPE T_Pile IS ACCESS T_Cellule; --T_Pile correspond au pointeur
                                   sur un élément
TYPE T_Cellule IS
  RECORD
    Valeur : Integer; --Ici on va créer une pile d'entiers
    Index : Integer; --Le dernier élément aura l'index N°1
    Suivant : T_Pile; --Le suivant correspond à une «sous-pile»
  END RECORD;
```

 La composante Index est faite pour pouvoir indexer les éléments de notre liste, les numérotter en quelques sortes. La «cellule» en bas de la pile (le premier élément ajouté) portera toujours le numéro 1. La cellule le plus au dessus de la pile aura donc un index correspondant au nombre d'éléments contenus dans la pile.

Création des primitives

L'idée, en créant ce package, est de concevoir un type T_Pile et toutes les primitives nécessaires à son utilisation de sorte que tout programmeur utilisant ce package n'ait plus besoin de connaître la structure du type T_Pile : il doit tout pouvoir faire avec les primitives fournies sans avoir à mettre les mains dans le cambouis (Nous verrons plus tard comment lui interdire tout accès à ce «cambouis» avec la **privatisation** des packages). De quelles primitives a-t-on alors besoin ? En voici une liste avec le terme anglais généralement utilisé et une petite explication :

- **Empiler (Push)** : ajoute un élément sur le dessus de la pile. Cet élément devient alors le premier et son indice (composante Index) est supérieur d'une unité à «l'ex-premier» élément.
- **Dépiler (Pop)** : retire le premier élément de la pile, la première cellule. Cette fonction renvoie en résultat la valeur de cette cellule.
- **Pile_Vide (Empty)** : renvoie **TRUE** si la pile est vide (c'est-à-dire **sipile = NULL**). Primitive importante car il est impossible de dépiler une pile vide.
- **Longueur (Length)** : renvoie la longueur de la pile, c'est à dire le nombre de cellules qu'elle contient.
- **Premier (First)** : renvoie la valeur de la première cellule de la pile sans la dépiler.

Bien, il ne nous reste plus qu'à créer ces primitives. Commençons par la procédure Push(P,n). Elle va recevoir une pile P, qui n'est rien d'autre qu'un pointeur vers la cellule située sur le dessus de la pile. P sera en mode **IN OUT**. Elle recevra également un Integer n (puisque nous avons choisi les Integer, mais nous pourrions tout aussi bien le faire avec des float ou autre chose). Elle va devoir créer une cellule (et donc un index), la faire pointer sur la cellule située sur le dessus de la pile avant d'être elle-même pointée par P. Compris ? Alors allez-y !

 Pensez bien à gérer tous les cas : «la pile P est vide» (P = **NULL**) et «la pile P n'est pas vide» ! Car je vous rappelle que si P = **NULL**, alors écrire «**P.all**» entraînera une erreur.

Secret (cliquez pour afficher)

Code : Ada

```
PROCEDURE Push (P : IN OUT T_Pile; N : IN Integer) IS
  Cell : T_Cellule;
BEGIN
  Cell.Valeur := N;
  IF P /= NULL
  THEN Cell.Index := P.all.Index + 1;
  Cell.Suivant := P.all'ACCESS;
  ELSE Cell.Index := 1;
  END IF;
  P := NEW T_Cellule'(Cell);
END Push;
```

Ne surtout pas écrire :

Code : Ada

```
P := Cell'access;
```

 Pourquoi ? Simplement parce que l'objet Cell a une durée de vie limitée à la procédure Push. Une fois cette dernière terminée, l'objet Cell de type T_Cellule disparaît et notre pointeur pointera alors sur... rien du tout !

Votre procédure Push() est créée ? Enregistrez votre package et créez un programme afin de la tester. Exemple :

Code : Ada

```
with ada.Text_IO, Ada.Integer_Text_IO, P_Pile;
use ada.Text_IO, Ada.Integer_Text_IO, P_Pile;

Procedure test is
  P : T_Pile;
BEGIN
  Push(P,3); Push(P,5); Push(P,1); Push(P,13);
  Put(P.all.Index); Put(P.all.Valeur);
  new_line;
  Put(P.Suivant.Index); Put(P.Suivant.Valeur);
  new_line;
  Put(P.Suivant.suivant.Index); Put(P.Suivant.suivant.Valeur);
  new_line;
  Put(P.Suivant.suivant.suivant.Index); Put(P.Suivant.suivant.suivant.Valeur);
  new_line;
  Put(P.Suivant.suivant.suivant.suivant.Index); Put(P.Suivant.suivant.suivant.suivant.Valeur);
```

```
end Test;
```

Maintenant, la procédure Pop(P,N). Je ne reviens pas sur les paramètres ou sur le sens de cette fonction, j'en ai déjà parlé. Je vous rappelle toutefois que cette procédure doit retourner, non pas un objet de type T_Cellule, mais une variable de type Integer, et non pas la pile. Les paramètres devront donc être en mode OUT (voir IN OUT). Dernier point, si la pile est vide, le programmeur qui utilisera votre package (appelons-le « programmeur final ») ne devrait pas la dépiler, mais il lui revient de le vérifier à l'aide de la primitive Empty : ce n'est pas le rôle de nos primitives de gérer les éventuelles bêtises du programmeur final. En revanche, c'est à vous de gérer le cas où la pile ne contiendrait qu'une seule valeur et deviendrait vide dans le code de votre primitive.

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
PROCEDURE pop(P : IN OUT T_Pile ; N : OUT Integer) IS
BEGIN
  N := P.all.valeur; --ou P.valeur
  --P.all est censé exister, ce sera au
programmeur final de le vérifier
  IF P.all.suivant /= NULL
  THEN P := P.suivant;
  ELSE P := null;
  END IF;
END Pop;
```

Maintenant que vos deux procédures principales sont créées (je vous laisse le soin de les tester) nous n'avons plus que trois primitives (assez simples) à coder : Empty(), First() et Length(). Toutes trois sont des fonctions ne prenant qu'un seul paramètre : une pile P. Elles retourneront respectivement un booléen pour la première et un integer pour les deux suivantes. Je vous écris ci-dessous le code source de P_Pile.adb et P_Pile.ads. Merci à qui ?

[Secret \(cliquez pour afficher\)](#)

Code : Ada - P_Pile.adb

```
PACKAGE BODY P_Pile IS
  PROCEDURE Push (
    P : IN OUT T_Pile;
    N : IN Integer) IS
    Cell : T_Cellule;
  BEGIN
    Cell.Valeur := N;
    IF P /= NULL
    THEN
      Cell.Index := P.all.Index + 1;
      Cell.Suivant := P.all'ACCESS;
    ELSE
      Cell.Index := 1;
    END IF;
    P := NEW T_Cellule'(Cell);
  END Push;

  PROCEDURE Pop (
    P : IN OUT T_Pile;
    N : OUT Integer) IS
  BEGIN
    N := P.all.Valeur; --ou P.valeur
    --P.all est censé exister, ce sera au programmeur final de
le vérifier
    IF P.all.Suivant /= NULL
    THEN
      P := P.Suivant;
    ELSE
      P := null;
    END IF;
  END Pop;

  FUNCTION Empty (
    P : IN T_Pile)
  RETURN Boolean IS
  BEGIN
    IF P = null
    THEN
      RETURN True;
    ELSE
      RETURN False;
    END IF;
  END Empty;

  FUNCTION Length(P : T_Pile) RETURN Integer IS
  BEGIN
    IF P = null
    THEN RETURN 0;
    ELSE RETURN P.Index;
    END IF;
  END Length;

  FUNCTION First(P : T_Pile) RETURN Integer IS
  BEGIN
    RETURN P.Valeur;
  END First;
END P_Pile;
```

Code : Ada - P_Pile.ads

```
PACKAGE P_Pile IS
  TYPE T_Cellule; --T_Cellule correspondant à
un élément de la pile
  TYPE T_Pile IS ACCESS ALL T_Cellule; --T_Pile correspondant au
pointeur sur un élément
  TYPE T_Cellule IS
  RECORD
    Valeur : Integer; --On crée une pile d'entiers
    Index : Integer;
    Suivant : T_Pile; --Le suivant correspond à une "sous-
pile"
  END RECORD;
  PROCEDURE Push (P : IN OUT T_Pile; N : IN Integer);
  PROCEDURE Pop (P : IN OUT T_Pile; N : OUT Integer);
  FUNCTION Empty (P : IN T_Pile) RETURN Boolean;
  FUNCTION Length(P : T_Pile) RETURN Integer;
  FUNCTION First(P : T_Pile) RETURN Integer;
END P_Pile;
```

Jouons avec le package P_Pile

Notre package est enfin fini. Notre TAD est prêt à l'emploi et l'impose maintenant une règle : interdiction de manipuler nos piles autrement qu'à l'aide de nos 5 primitives ! Donc, plus de P.valeur, seulement des First(P)! Plus de P.suivant, il faudra faire avec Pop(P)! Cela va quelque peu compliquer les choses mais nous permettra de comprendre comment se manipulent les TAD sans pour autant partir dans des programmes de 300 lignes. L'objectif avec notre type T_Pile, sera de créer une procédure Put(p : T_Pile) qui se charge d'afficher une pile sous forme de colonne. Attention, le but de ce cours n'est pas de créer toutes les fonctions ou procédures utiles à tous les TAD. Il serait par exemple intéressant de créer une procédure ou une fonction get(P). Mais nous n'allons nous concentrer que sur une seule procédure à chaque fois. Libre à vous de combler les manques.

Concernant notre procédure Put(P), deux possibilités s'offrent à nous : soit nous employons une boucle itérative (LOOP, FOR, WHILE), soit nous employons une boucle récursive. Vous allez vous rendre compte que la récursivité siéderait parfaitement aux TAD (ce sera donc ma première solution), mais je vous proposerai également deux autres façons de procéder (avec une boucle FOR et une boucle WHILE). Voici le premier code, avec la méthode récursive :

Code : Ada

```

procedure Put(P : T_Pile) is
    Pile : T_Pile := P;
    N : integer;
begin
    if not Empty(Pile)
    then
        Put(N);
        new_line;
        Put(Pile);
    end if;
end Put;

```

Regardons un peu ce code. Tout d'abord, c'est un code que j'aurais tout à fait pu vous proposer avant le chapitre sur les pointeurs : qui peut deviner en voyant ces lignes comment est constitué le type T_Pile ? Personne, car nous avons fourni tous les moyens pour ne pas avoir besoin de le savoir (c'est ce que l'on appelle l'**encapsulation**, encore une notion que nous verrons dans la partie IV). Que fait donc ce code ? Tout d'abord il teste si la pile est vide ou non. Ce sera toujours le cas. Si elle est vide, c'est fini ; sinon, on dépile la pile afin de pouvoir afficher son premier élément puis on demande d'afficher la nouvelle pile, privée de sa première cellule. J'aurais pu également écrire *Put(First(Pile))* avant de dépiler, mais cela n'aurait contraint à manipuler deux fois la pile sans pour autant économiser de variable.

 D'ailleurs, tu aurais aussi pu économiser une variable ! Ton objet Pile n'a aucun intérêt, il suffit de dépiler P et d'afficher P !

Non ! Surtout pas ! Tout d'abord, je vous rappelle que P est obligatoirement en mode **IN** puisqu'il est transmis à la fonction *Empty()*. Or, la procédure *pop()* a besoin d'un paramètre en mode **OUT** pour pouvoir dépiler. Il y aurait un conflit. Ensuite, en supportant que le compilateur ne bronche pas, que deviendrait P après son affichage ? Eh bien P serait vide, on aurait perdu toutes les valeurs qui la composait, simplement pour un affichage. C'est pourquoi j'ai créé un objet *Pile* qui sert ainsi de «roue de secours». Voici maintenant une solution itérative :

Code : Ada

```

while not Empty(Pile) loop
    put(First(P));
    pop(Pile, N);
    new_line;
end loop;

```

On répète ainsi les opérations tant que la liste n'est pas vide : afficher le premier élément, dépiler. Enfin, voici une seconde solution itérative (dernière solution proposée) :

Code : Ada

```

for i in 1..Length(P) loop
    put(first(Pile));
    pop(Pile,N);
    new_line;
end loop;

```

Ce qui est utilisé ici, ce n'est pas le fait que la liste soit vide, mais le fait que sa longueur soit nulle. Les opérations sont toutefois toujours les mêmes.

Les files

Implémentation

Deuxième exemple, les files. Nous allons donc créer un nouveau package *P_File*. La différence avec la pile ne se fait pas au niveau de l'implémentation. Les files restent linéaires, non cycliques, simplement chaînées et non contraintes.

Code : Ada

```

TYPE T_Cellule;
TYPE T_File IS ACCESS ALL T_Cellule;
TYPE T_Cellule IS
  RECORD
    Valeur : Integer;
    Index : Integer;
    Suivant : T_File;
  END RECORD;

```

La différence se fait donc au niveau des primitives puisque c'est la «méthode d'ajout» qui diffère. Les files sont gérées en FIFO : first in, first out. Imaginez pour cela une file d'attente à une caisse : tout nouveau client (une cellule) doit se placer à la fin de la file, la caissière (le programmeur final ou le programme) ne peut traiter que le client en début de file, les autres devant attendre leur tour. Alors de quelles primitives a-t-on besoin ?

- **Enfiler (Enqueue)** : ajoute un élément à la fin de la file. Son indice est donc supérieur d'une unité à l'ancien dernier élément.
- **Défiler (Dequeue)** : renvoie le premier élément de la file et le retire. Cela implique de renuméroter toute la file.
- **File_Vide (Empty)** ; **Longueur (Length)** ; **Premier (First)** : même sens et même signification que pour les piles.

Notre travail consistera donc à modifier notre package *P_Pile* en conséquence. Attention à ne pas aller trop vite ! Les changements, même s'ils semblent à priori mineurs, ne portent pas seulement sur les noms des primitives. L'ajout d'une cellule notamment, n'est plus aussi simple qu'àuparavant. Ce travail étant relativement simple (pour peu que vous le preniez au sérieux), il est impératif que vous le fassiez par vous-même ! Cela constituera un excellent exercice sur les pointeurs et la récursivité. Ne consultez mon code qu'à titre de correction. C'est important avant d'attaquer des TAD plus complexes.

Secret (cliquez pour afficher)

Code : Ada - P_File.adb

```

PACKAGE BODY P_File IS
  PROCEDURE Enqueue (F : IN OUT T_File ; N : IN      Integer) IS
    Cell : T_Cellule;
  BEGIN
    Cell.Valeur := N;
    Cell.Suivant := NULL;
    IF F = NULL
    then
      Cell.Index := 1;
      F := NEW T_Cellule'(Cell);
    ELSEIF F.Suivant = NULL
    then
      Cell.Index := F.Index + 1;
      F.Suivant := NEW T_Cellule'(Cell);
    ELSE
      Enqueue(F.Suivant, N);
    END IF;
    END Enqueue;

  PROCEDURE Dequeue (F : IN OUT T_File ; N :      OUT Integer) IS
    procedure Decrement(F : IN T_File) is
    begin
      if F /= null
      then
        F.Index := F.index - 1;
        Decrement(F.suivant);
      end if;
    end Decrement;

    BEGIN
      N := F.Valeur;
      IF F.Suivant /= NULL
      THEN
        Decrement(F);
      ELSE
        F := NULL;
      END IF;
    END Dequeue;

  FUNCTION Empty (F : IN      T_File) RETURN Boolean IS
  BEGIN
    IF F=NULL
    THEN RETURN True;
    ELSE RETURN False;
  END;

```

```

    END IF ;
END Empty ;

FUNCTION Length ( F : T_File ) RETURN Integer IS
BEGIN
  IF F = NULL
    THEN RETURN 0 ;
  ELSIF F.Suivant = NULL
    THEN RETURN F.Index ;
  ELSE RETURN Length(F.Suivant) ;
  END IF ;
END Length ;

FUNCTION First ( F : T_File ) RETURN Integer IS
BEGIN
  RETURN F.Valeur ;
END First ;

END P_File;

```

Code : Ada - P_File.ads

```

PACKAGE P_Pile IS
  TYPE T_Cellule;           --T_Cellule correspondant à
  un élément de la pile
  TYPE T_Pile IS ACCESS ALL T_Cellule;--T_Pile correspondant au
  pointeur sur un élément

  TYPE T_Cellule IS
  RECORD
    Valeur : Integer; --On crée une pile d'entiers
    Index : Integer;
    Suivant : T_Pile; --Le suivant correspond à une "sous-
  pile"
  END RECORD;

  PROCEDURE Push (P : IN OUT T_Pile; N : IN Integer);
  PROCEDURE Pop (P : IN OUT T_Pile ; N : OUT Integer);
  FUNCTION Empty (P : IN T_Pile) RETURN Boolean;
  FUNCTION Length(P : T_Pile) RETURN Integer;
  FUNCTION First(P : T_Pile) RETURN Integer;
END P_Pile;

```

La procédure enqueue a besoin d'être récursive car on ajoute une cellule à la fin de la liste et non au début ! La procédure dequeue doit non seulement supprimer le premier élément de la file, mais également décrémente tous les indices, sans quoi le premier sera le n°2 !!! Enfin, la fonction Length doit elle aussi être récursive pour lire l'indice de la dernière cellule.

Amusons-nous encore

Comme promis, nous allons désormais tester notre structure de données. Mais hors de question de se limiter à refaire une procédure Put() ! Ce serait trop simple. Cette fois nous allons également créer une fonction de **concaténation** !

Roh ! Encore de la théorie ! Il a pas fini avec ses termes compliqués ? 😊

Pas d'emballage ! La concaténation est une chose toute simple ! Cela consiste juste à mettre bout à bout deux listes pour en faire une plus grande ! Le symbole régulièrement utilisé en Ada (et dans de nombreux langages) est le symbole & (le ET commercial, aussi appelé esperluette). Vous vous souvenez ? Nous l'avions vu lors du chapitre sur les [chaînes de caractères](#). Donc à vous de créer cette fonction de sorte que l'on puisse écrire : File1 & File 3 ! N'oubliez pas auparavant de créer une procédure Put() pour contrôler vos résultats et prenez garde que l'affichage d'une file ne la défile pas définitivement !

Secret (cliquez pour afficher)

Code : Ada

```

procedure Put(F : in out T_File) is
  N : integer;
begin
  for i in 1..length(F) loop
    dequeuer(F,N);
    put(N);
    enqueue(F,N); --on n'oublie pas de réenfiler
    l'élément N pour ne pas vider notre file !
  end loop;
end Put;

function "&"(left,right : in T_File) return T_File is
  N : integer;
  L : T_File := Left;
  R : T_File := Right;
  res : T_File;
begin
  for i in 1..length(left) loop --on place les éléments de la
  file de gauche dans la file résultat
    dequeuer(L,N);
    enqueue(res,N);
    enqueue(L,N);
  end loop;
  for i in 1..length(right) loop --puis on place ceux de la
  file de droite dans la file résultat
    dequeuer(R,N);
    enqueue(res,N);
    enqueue(R,N);
  end loop;
  return res;
end;

```

Cette fois, j'ai opté pour des algorithmes itératifs. Après un rapide essai, on se rend compte que les algorithmes récursifs ont tendance à inverser nos listes.

Explication : si vous avez une liste (3;6;9) et que vous employez un algorithme récursif vous allez d'abord défilé le 3 pour l'afficher et avant de le renfiler, vous allez relancer votre algorithme, gardant le 3 en mémoire. Donc vous défilé le 6 pour l'afficher et avant de le renfiler vous allez relancer votre algorithme, gardant le 6 en mémoire. Enfin, vous défilé le 9 pour l'afficher, la liste étant vide désormais vous réenfilerez le 9, puis le 6, puis le 3. Au final, un simple affichage fait que vous vous retrouvez non plus avec une liste (3;6;9) mais (9;6;3). Ce problème peut être résolu en employant un sous-programme qui inversera la file, mais cette façon de procéder vous semble-t-il naturelle ? Pas à moi toujours. 😊

Les listes chaînées Quelques rappels

Comme vous vous en êtes sûrement rendu compte en testant vos fonctions et procédures, l'utilisation des files, si elle semble similaire à celle des piles, est en réalité fort différente. Chacun de ces TAD a ses avantages et inconvénients. Mais le principal défaut de ces TAD est d'être obligé de défiler/dépiler les éléments les uns après les autres pour pouvoir les traiter, ce qui oblige à les remplir/renfiler la suite avec tous les risques que cela implique quant à l'ordre des éléments. Notre troisième TAD va donc régler ce souci puisque nous allons traiter des listes chaînées (et ce sera le dernier TAD linéaire que nous traiterons).

Comme dit précédemment, les listes chaînées sont linéaires, doubllement chaînées, non cycliques et non contraintes. Qui plus est, elles pourront être traitées en LIFO (comme une pile) ou en FIFO (comme une file) mais il sera également possible d'insérer un élément au beau «milieu» de la liste. Pour cela, il devra être possible de parcourir la liste sans la «démontter» comme nous le faisons avec les piles ou les files.

Encore un nouveau package ? Je commence à me lasser, moi 😊

Heureusement pour vous, le langage Ada intègre, depuis la norme 2005, de nouveaux packages appelés Ada.Containers et qui nous fournissent quelques TAD comme les tables de hachage mais surtout les Doubly_Linked_List (listes doubllement chaînées) et les Vectors (vecteurs). Ces deux derniers types sont relativement similaires et correspondent aux listes chaînées. Leur différence ? Les Vectors se comportent un peu à la manière des tableaux, chaque vector étant indexé (en général, indexé avec des entiers naturels commençant à 0 ou 1) de sorte qu'il suffit de disposer du numéro d'indice pour accéder à l'élément voulu, alors que les Doubly_Linked_Lists n'utilisent pas d'indices mais un curseur qui pointe sur un élément de la liste et que l'on peut

déplacer vers le début ou la fin de la liste.

Le package Ada.Containers.Doubly_Linked_Lists

Mise en place

Commençons par évoquer le type List. Il est accessible via le package Ada.Containers.Doubly_Linked_Lists. Pour lire les spécifications de ce package, il vous suffit d'ouvrir le fichier appelé a-dll.adidl.situation dans le répertoire d'installation de GNAT. Vous trouverez normalement ce répertoire à l'adresse C:\GNAT sous Windows. Il faudra ensuite ouvrir les dossiers 2011\lib\gcc\i686-pc-mingw32\4.5.3\adainclude (ou quelque chose de ce style, les dénominations pouvant varier d'une version à l'autre).

Comme le package Ada.Numerics.Discrete.Random qui nous servait à générer des nombres aléatoires, le package Ada.Containers.Doubly_Linked_Lists est un package générique (nous verrons bientôt la généréricité, patience), fait pour n'importe quel type de donnée et nous ne pouvons donc pas l'utiliser en état. Nous devons tout d'abord **instancier**, c'est-à-dire créer un sous-package prévu pour le type de données désiré. Exemple en image :

Code : Ada

```
with Ada.Containers.Doubly_Linked_Lists;
type T_Score is record
    name : string(1..3) := " ";
    value : natural := 0;
end record;
package P_Lists is new Ada.Containers.Doubly_Linked_Lists(T_Score);
use P_Lists;
L : List;
C : Cursor;
```

Primitives

Nous avons ainsi créé un package P_Lists nous permettant d'utiliser des listes chaînées contenant des éléments de type T_Score. Une liste chaînée est de type List (sans S à la fin !). Nous disposons des primitives suivantes pour comparer deux listes, connaître la longueur d'une liste, savoir si elle est vide, la vider, ajouter un élément au début (Prepend) ou l'ajouter à la fin (Append) :

Code : Ada

```
function "=" (Left, Right : List) return Boolean;           -- comparaison de deux listes
function Length (Container : List) return Count_Type;      -- longueur de la liste
function Is_Empty (Container : List) return Boolean;        -- la liste est-elle vide ?
procedure Clear (Container : in out List);                -- vide la liste
procedure Prepend (Container : in out List;               -- ajoute un élément en début de liste
                   New_Item : Element_Type;
                   Count   : Count_Type := 1);          -- ajoute
procedure Append (Container : in out List;               -- ajoute un élément en fin de liste
                  New_Item : Element_Type;
                  Count   : Count_Type := 1);          -- ajoute
```

De même, il est possible de connaître le premier ou le dernier élément (sans le « délier »), de le supprimer, de savoir si la liste contient un certain élément ou pas ou encore d'inverser la liste à l'aide des primitives suivante :

Code : Ada

```
function First_Element (Container : List) return Element_Type; -- renvoie le premier élément
function Last_Element (Container : List) return Element_Type;   -- renvoie le dernier élément
procedure Delete_First;                                         -- supprime le premier élément
    (Container : in out List;
     Count   : Count_Type := 1);
procedure Delete_Last;                                         -- supprime le dernier élément
    (Container : in out List;
     Count   : Count_Type := 1);
function Contains;                                              -- indique si la liste contient cet élément
    (Container : List;
     Item    : Element_Type) return Boolean;
procedure Reverse_Elements (Container : in out List);           -- renverse la liste
```



Ce curseur est là pour nous situer sur la liste, pour pointer un élément. Nous ne pouvons travailler seulement avec le premier et le dernier élément. Supposons que nous ayons la liste suivante :

Name : mac Value : 500	Name : mic Value : 0	Name : bob Value : 1800	Name : joe Value : 5300	Name : mac Value : 800
Liste de scores et curseur				

Le curseur indique le second élément de la liste. Dès lors, nous pouvons lire cet élément, le modifier, insérer un nouvel élément juste avant lui, le supprimer ou bien l'échanger avec un élément pointé par un second curseur avec les primitives suivantes :

Code : Ada

```
function Element (Position : Cursor) return Element_Type; -- renvoie l'élément pointé
procedure Replace_Element;                                     -- modifie l'élément pointé
    (Container : in out List;
     Position : Cursor;
     New_Item : Element_Type);
procedure Insert;                                              -- insère un élément devant l'élément pointé
    (Container : in out List;
     Before  : Cursor;
     New_Item : Element_Type;
     Count   : Count_Type := 1);
procedure Delete;                                              -- supprime l'élément pointé
    (Container : in out List;
     Position : in out Cursor;
     Count   : Count_Type := 1);
procedure Swap;                                                 -- échange deux éléments pointés
    (Container : in out List;
     I, J      : Cursor);
```

Ce curseur peut être placé au début, à la fin de la liste ou être déplacé élément après élément.

Code : Ada

```
function First (Container : List) return Cursor; --place le
cursor au début
function Last (Container : List) return Cursor; --place le
cursor à la fin
function Next (Position : Cursor) return Cursor; --déplace le
cursor sur l'élément suivant
procedure Next (Position : in out Cursor);
function Previous (Position : Cursor) return Cursor; --déplace le
cursor sur l'élément précédent
procedure Previous (Position : in out Cursor);
```

Enfin, il est possible d'appliquer une procédure à tous les éléments d'une liste en utilisant un pointeur sur procédure avec :

Code : Ada

```
procedure Iterate(Container : List; Process : not null access
procedure (Position : Cursor));
```

Une application

Nous allons créer un programme créant une liste de 5 scores et les affichant. Je vais vous proposer deux procédures d'affichage distinctes. Mais tout d'abord, créons notre liste :

Code : Ada

```
Append(L, ("mac",500)) ;
Append(L, ("mic",0)) ;
Append(L, ("bob",1800)) ;
Append(L, ("joe",5300)) ;
Append(L, ("mac",800)) ;
```

Vous avez remarqué, j'ai créé la même liste que tout à l'heure, avec la procédure Append. Une autre façon de procéder, mais avec la procédure Prepend donnerait cela :

Code : Ada

```
Prepend(L, ("mac",800)) ;
Prepend(L, ("joe",5300)) ;
Prepend(L, ("bob",1800)) ;
Prepend(L, ("mic",0)) ;
Prepend(L, ("mac",500)) ;
```

Une dernière méthode (un peu tirée par les cheveux), avec le curseur donnerait :

Code : Ada

```
Append(L, ("joe",5300)) ;
C := first(L) ;
insert(L,C,("mac",500)) ;
insert(L, ("bob",1800)) ;
previous(C) ;
insert(L, ("mic",0)) ;
Append("mac",500) ;
```

 Les appels Append(L, ("mac", 500)) peuvent également s'écrire L.Append(("mac", 500)). Nous verrons bientôt pourquoi.

Enfin, voici une première façon, laborieuse d'afficher les éléments de la liste :

Code : Ada

```
procedure put(score : T_Score) is
begin
  Score := Element(C) ;
  put_line(Score.name & " a obtenu " & integer'image(Score.value) &
  " points.") ;
end put ;
...
begin
  C := first(L) ;
  for i in 1..length(L)-1 loop
    put(Element(c)) ;
    next(c) ;
  end loop ;
  put(Element(c)) ;
end ;
```

Cette première façon utilise le curseur et passe en revue chacun des éléments les uns après les autres. Attention toutefois au cas du dernier élément qui n'a pas de successeur ! A noter également qu'une procédure récursive aurait aussi bien pu faire l'affaire. Enfin, voici une seconde façon de procéder utilisant les pointeurs sur procédure cette fois :

Code : Ada

```
procedure put(C : Cursor) is
  score : T_Score;
begin
  Score := Element(C) ;
  put_line(Score.name & " a obtenu " & integer'image(Score.value) &
  " points.") ;
end put ;
...
begin
  iterate(L,Put'access) ;
end ;
```

En utilisant la procédure iterate(), nous nous épargnons ici un temps considérable. Cela justifie amplement les efforts que vous avez fourni pour comprendre ce qu'étaient les pointeurs sur procédure.



L'appel iterate(L, Put'access) peut être remplacé par L.iterate(put'access) pour plus de lisibilité !

Encore une fois, une explication sera fournie très prochainement dans la partie IV.

Le package Ada.Containers.Vectors

Mise en place

Évoquons maintenant le type vector. Il est accessible via le package Ada.Containers.Vectors. Pour lire les spécifications de ce package, il vous suffit d'ouvrir le fichier appelé a-convec.adb. Encore une fois, ce package doit être instancié de la manière suivante :

Code : Ada

```
with Ada.Containers.Vectors ;
package P_Vectors is new Ada.Containers.Vectors(Positive,T_Score) ;
use P_Vectors ;
```

V : Vector ;

Nous créons ainsi un package P_Vectors indexé à l'aide du type Positive (nombre entiers strictement positifs) et contenant des éléments de type T_Score. Nous aurions pu utiliser le type Natural pour l'indexation, mais le type Positive nous garantit que le premier élément du Vector V sera le numéro 1 et non le numéro 0.

 Il est possible d'utiliser également un curseur avec les Vectors, mais ce n'est pas la solution la plus évidente : mieux vaut nous servir des indices.

Primitives

Les primitives disponibles avec les Doubly_Linked_Lists sont également disponibles pour les Vectors :

Secret (cliquez pour afficher)

Code : Ada

```
overriding function "=" (Left, Right : Vector) return Boolean; --  
test d'égalité entre deux vectors  
function Length (Container : Vector) return Count_Type; --  
renvoie la longueur du vector  
function Is_Empty (Container : Vector) return Boolean; --  
renvoie true si le vector est vide  
procedure Clear (Container : in out Vector); --  
vide le vector  
procedure Prepend --  
ajoute un élément au début du vector  
(Container : in out Vector;  
New_Item : Vector);  
procedure Append --  
ajoute un élément à la fin du vector  
(Container : in out Vector;  
New_Item : Vector);  
function First_Element (Container : Vector) return Element_Type; --  
-renvoie la valeur du premier élément du vector  
function Last_Element (Container : Vector) return Element_Type; --  
-renvoie la valeur du dernier élément du vector  
procedure Delete_First --  
-supprime le premier élément du vector  
(Container : in out Vector;  
Count : Count_Type := 1);  
procedure Delete_Last --  
-supprime le dernier élément du vector  
(Container : in out Vector;  
Count : Count_Type := 1);  
function Contains --  
-indique si un élément est présent dans le vector  
(Container : Vector;  
Item : Element_Type) return Boolean;  
procedure Reverse_Elements (Container : in out Vector); --  
-renverse l'ordre des éléments du vector
```

D'autres primitives font appel à un curseur, mais je n'y ferai pas référence (vous pouvez les découvrir par vous même). Mais elles ont en général leur équivalent, sans emploi du curseur mais avec un indice.

Secret (cliquez pour afficher)

Code : Ada

```
function Element --renvoie la  
valeur de l'élément situé à l'indice spécifié  
(Container : Vector;  
Index : Index_Type) return Element_Type;  
procedure Replace_Element --supprime  
l'élément situé à l'indice spécifié  
(Container : in out Vector;  
Index : Index_Type;  
New_Item : Element_Type);  
procedure Insert --insère un  
nouvel élément avant l'indice spécifié  
(Container : in out Vector;  
Before : Extended_Index;  
New_Item : Element_Type;  
Count : Count_Type := 1);  
procedure Delete --supprime  
l'élément situé à l'indice spécifié  
(Container : in out Vector;  
Index : Extended_Index;  
Count : Count_Type := 1);  
procedure Swap (Container : in out Vector; I, J : Index_Type); --  
échange deux éléments  
function First_Index (Container : Vector) return Index_Type; --  
renvoie l'indice du premier élément  
function Last_Index (Container : Vector) return Extended_Index; --  
renvoie l'indice du dernier élément  
procedure Iterate --applique la  
procédure Process à tous les éléments du Vector  
(Container : Vector;  
Process : not null access procedure (Position : Cursor));
```

À ces primitives similaires à celles des listes doublement chaînées, il faut encore en ajouter quelques autres :

Code : Ada

```
--Fonctions de concaténation  
function "&" (Left, Right : Vector) return Vector;  
function "&" (Left : Vector; Right : Element_Type) return Vector;  
function "&" (Left : Element_Type; Right : Vector) return Vector;  
function "&" (Left, Right : Element_Type) return Vector;  
  
procedure Set_Length --crée un vecteur de longueur  
donnée  
(Container : in out Vector;  
Length : Count_Type);  
procedure Insert_Space --insère un élément vide avant  
l'indice spécifié  
(Container : in out Vector;  
Before : Extended_Index;  
Count : Count_Type := 1);  
function Find_Index --trouve l'indice d'un élément  
dans un tableau  
(Container : Vector;  
Item : Element_Type;  
Index : Index_Type := Index_Type'First) return  
Extended_Index;
```

Bien sûr, je ne vous ai pas dressé la liste de toutes les fonctions et procédures disponibles, mais vous avez ainsi les principales. À savoir : le type Element_Type correspond au type avec lequel vous aurez instancié votre package. De plus, le type Count est une sorte de type integer.

Nous en avons fini avec ce cours semi-théorique, semi-pratique. Nous n'avons bien sûr pas fait le tour de tous les TAD, mais il nous est pas interdit (bien au contraire) de développer vos propres types comme des arbres ou des cycles. Cela constitue un excellent exercice d'entraînement à l'algorithme et à la programmation qui vous poussera à manipuler les pointeurs et à user de programmes récursifs.

En résumé :

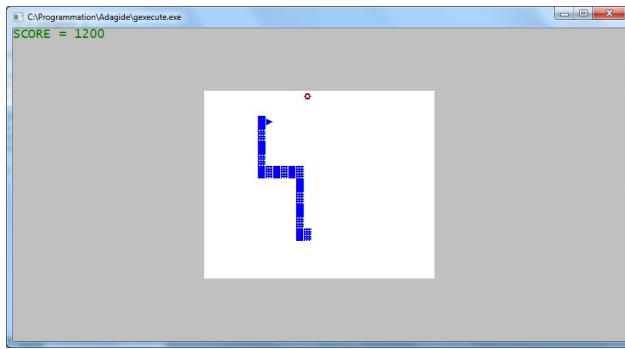
- Les TAD ont l'avantage de fournir des conteneurs de taille inconnue et donc non contraints, contrairement aux tableaux. Leur emploi est certes moins évident jusqu'à ce qu'ils soient créés vous-même ou faire appel à un package générique, mais ils résoudront bon nombre de problèmes. Pensez à l'historique de votre navigateur internet, à la liste des items dans un jeu vidéo ou plus simplement aux Unbounded_Strings.
- Si les TAD peuvent très bien être traités par une boucle itérative (**LOOP**, **FOR** ou **WHILE**), leur structure chaînée et leur longueur potentiellement très grande en font des sujets de choix pour les programmes récursifs. Et le besoin de récursivité ne cessera d'augmenter avec la complexité de la structure : imaginez seulement la difficulté de traiter un arbre avec des boucles itératives.

- N'oubliez pas que ces structures ne sont rendues possibles que par l'usage intensif des pointeurs. Si les packages Ada.Container ont été codés avec sérieux, cela ne doit pas vous en faire oublier les risques : évitez, par exemple, d'accéder à un élément d'une liste vide ou de lui retirer un élément !

[TP] Le jeu du serpent

La troisième partie se termine (*«Enfin !»* diront certains) et nous allons donc la conclure comme il se doit par un troisième TP ! Après avoir hésité entre la conception de divers programmes, tel un éditeur de carte ou un éditeur de texte, je me suis dit qu'après le difficile TP précédent et cette longue partie III, vous aviez bien mérité un TP moins décourageant et plus ludique. C'est pourquoi, après de longues réflexions, j'ai opté pour un jeu : le jeu du Serpent. Vous savez ce vieux jeu que l'on avait sur les vieux portables où un serpent devait manger sans cesse de gros pixels afin de grandir en prenant garde de ne jamais dévorer sa propre queue ou sortir de l'écran.

Comme d'habitude, je vais vous guider dans sa mise en œuvre, et comme d'habitude tout se fera en console mais... avec des **couleurs** ! Prêts ? Voici une image pour vous mettre en bouche :



Cahier des charges

Fonctionnalités

Comme toujours, avant de nous lancer à l'aveuglette, nous devons définir les fonctionnalités de notre programme, les limites du projet. Il n'est pas question de créer un jeu en 3D ou avec des boutons, des images, une base de données... nous allons faire plus sobre.

Notre programme devra :

- s'exécuter en console, je vous l'ai dit. Toutefois, nous ferons en sorte d'afficher quelques couleurs pour égayer tout ça et surtout rendre notre programme facilement jouable. Rassurez-vous, je vous fournirai un package pour cela;
- se jouer en temps réel, pas au tour par tour ! Si le joueur ne touche pas au clavier alors son serpent ira droit dans le mur;
- gérer et donc éviter les bogues : serpent faisant demi-tour sur lui-même, sortant de la carte ou dont le corps ne suivrait pas la tête dans certaines configurations (je ne rigole pas, vous commettrez sûrement ce genre d'erreur);
- permettre au joueur de diriger le serpent à l'aide des touches fléchées.

Organisation des types et variables

Nous ne savons pas quand le joueur perdra, notre serpent devra donc pouvoir s'allonger indéfiniment (en théorie). Il serait donc judicieux d'utiliser les TAD vu précédemment pour enregistrer les différents anneaux de notre reptile : il sera plus aisément d'agrandir notre serpent à l'aide des primitives append ou prepend qu'en déclarant des tableaux ou je ne sais quoi d'autre. De même, déplacer le serpent reviendrait simplement à retirer le dernier anneau de son corps pour le placer en premier dans une nouvelle position. Vous avez le choix entre tous les TAD que vous voulez, mais les Doubly_Linked_Lists ou les Vectors sont les mieux adaptés et comme j'ai une préférence pour le type Vector, la solution que je vous fournirai utilise donc... les Doubly_Linked_Lists (cherchez la cohérence ☺).

Mais que contiendra notre liste ? Elle contiendra toute une suite de coordonnées : les coordonnées des différentes parties du corps du serpent. Mais ce n'est pas tout ! Le serpent ne peut se limiter à une liste de coordonnées, il faudra également que notre type T_Serpent confirme la direction de la tête du serpent.

Enfin, la taille de l'aire de jeu, la vitesse de déplacement du serpent ou les couleurs utilisées devraient être enregistrées dans des variables (ou des constantes si besoin) toutes inventoriées dans un package. En faisant cela, ces variables deviendront des **variables globales**, ce qui est généralement risqué mais clarifiera notre code et simplifiera toute modification des paramètres du jeu. Nous reviendrons sur les variables globales dans la prochaine partie.

Un package bien utile

Le package NT_Console

Depuis le début je vous dis que notre programme sera en couleur, donc il est temps de vous fournir le package nécessaire pour utiliser des couleurs dans la console. Il ne s'agit ni d'un package officiel, ni d'un package de mon cru (*mea maxima culpa* ☺) mais d'un vieux package mis à notre disposition par Jerry van Dijk et libre de diffusion, appelé NT_Console :

```
Secret (cliquez pour afficher)
Code : Ada - NT_Console.adb

-----  
--  
-- File: nt_console.adb  
-- Description: Win95/NT console support  
-- Rev: 0.3  
-- Date: 08-june-1999  
-- Author: Jerry van Dijk  
-- Mail: jdijk@acm.org  
--  
-- Copyright (c) Jerry van Dijk, 1997, 1998, 1999  
-- Billie Holidaystraat 28  
-- 2324 LK LEIDEN  
-- THE NETHERLANDS  
-- tel int + 31 71 531 43 65  
--  
-- Permission granted to use for any purpose, provided this  
copyright  
-- remains attached and unmodified.  
--  
-- THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR  
-- IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED  
-- WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR  
PURPOSE.  
--  
-----  
  
pragma C_Pass_By_Copy (128);  
with Interfaces; use Interfaces;  
package body NT_Console is  
  pragma Linker_Options (" -luser32");  
  -----  
  -- WIN32 INTERFACE --  
    
  Beep_Error : exception;  
  Fill_Char_Error : exception;  
  Cursor_Get_Error : exception;  
  Cursor_Set_Error : exception;  
  Cursor_Pos_Error : exception;  
  Buffer_Info_Error : exception;  
  Set_Attribute_Error : exception;  
  Invalid_Handle_Error : exception;  
  Fill_Attribute_Error : exception;  
  Cursor_Position_Error : exception;  
    
  subtype DWORD is Unsigned_32;  
  subtype HANDLE is Unsigned_32;
```

```

subtype WORD is Unsigned_16;
subtype SHORT is Short_Integer;
subtype WIBOOL is Integer;

type LPDWORD is access all DWORD;
pragma Convention (C, LPDWORD);

type Nibble is mod 2 ** 4;
for Nibble'Size use 4;

type Attribute is
  record
    Foreground : Nibble;
    Background : Nibble;
    Reserved : Unsigned_8 := 0;
  end record;

for Attribute'Size use 16;
pragma Convention (C, Attribute);

type COORD is
  record
    X : SHORT;
    Y : SHORT;
  end record;
pragma Convention (C, COORD);

type SMALL_RECT is
  record
    Left : SHORT;
    Top : SHORT;
    Right : SHORT;
    Bottom : SHORT;
  end record;
pragma Convention (C, SMALL_RECT);

type CONSOLE_SCREEN_BUFFER_INFO is
  record
    Size : COORD;
    Cursor_Pos : COORD;
    Attrib : Attribute;
    Window : SMALL_RECT;
    Max_Size : COORD;
  end record;
pragma Convention (C, CONSOLE_SCREEN_BUFFER_INFO);

type PCONSOLE_SCREEN_BUFFER_INFO is access all
CONSOLE_SCREEN_BUFFER_INFO;
pragma Convention (C, PCONSOLE_SCREEN_BUFFER_INFO);

type CONSOLE_CURSOR_INFO is
  record
    Size : DWORD;
    Visible : WIBOOL;
  end record;
pragma Convention (C, CONSOLE_CURSOR_INFO);

type PCONSOLE_CURSOR_INFO is access all CONSOLE_CURSOR_INFO;
pragma Convention (C, PCONSOLE_CURSOR_INFO);

function GetCh return Integer;
pragma Import (C, GetCh, "_getch");

function KbHit return Integer;
pragma Import (C, KbHit, "_kbhit");

function MessageBeep (Kind : DWORD) return DWORD;
pragma Import (StdCall, MessageBeep, "MessageBeep");

function GetStdHandle (Value : DWORD) return HANDLE;
pragma Import (StdCall, GetStdHandle, "GetStdHandle");

function GetConsoleCursorInfo (Buffer : HANDLE; Cursor : PCONSOLE_CURSOR_INFO) return WIBOOL;
pragma Import (StdCall, GetConsoleCursorInfo,
"GetConsoleCursorInfo");

function SetConsoleCursorInfo (Buffer : HANDLE; Cursor : PCONSOLE_CURSOR_INFO) return WIBOOL;
pragma Import (StdCall, SetConsoleCursorInfo,
"SetConsoleCursorInfo");

function SetConsoleCursorPosition (Buffer : HANDLE; Pos : COORD) return DWORD;
pragma Import (StdCall, SetConsoleCursorPosition,
"SetConsoleCursorPosition");

function SetConsoleTextAttribute (Buffer : HANDLE; Attr : Attribute) return DWORD;
pragma Import (StdCall, SetConsoleTextAttribute,
"SetConsoleTextAttribute");

function GetConsoleScreenBufferInfo (Buffer : HANDLE; Info : PCONSOLE_SCREEN_BUFFER_INFO) return DWORD;
pragma Import (StdCall, GetConsoleScreenBufferInfo,
"GetConsoleScreenBufferInfo");

function FillConsoleOutputCharacter (Console : HANDLE; Char : Character; Length : DWORD; Start : COORD; Written : LPDWORD) return DWORD;
pragma Import (Stdcall, FillConsoleOutputCharacter,
"FillConsoleOutputCharacterA");

function FillConsoleOutputAttribute (Console : Handle; Attr : Attribute; Length : DWORD; Start : COORD; Written : LPDWORD) return DWORD;
pragma Import (Stdcall, FillConsoleOutputAttribute,
"FillConsoleOutputAttribute");

WIN32_ERROR : constant DWORD := 0;
INVALID_HANDLE_VALUE : constant HANDLE := -1;
STD_OUTPUT_HANDLE : constant DWORD := -11;

Color_Value : constant array (Color_Type) of Nibble := (0,
1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15);
Color_Type_Value : constant array (Nibble) of Color_Type :=
(Black, Blue, Green, Cyan, Red, Magenta, Brown, Gray,
Black, Light_Blue, Light_Green, Light_Cyan, Light_Red,
Light_Magenta, Yellow, White);

-- PACKAGE VARIABLES --
-----

Output_Buffer : HANDLE;
Num_Bytes : aliased DWORD;
Num_Bytes_Access : LPDWORD := Num_Bytes'Access;
Buffer_Info_Rec : aliased CONSOLE_SCREEN_BUFFER_INFO;
Buffer_Info : PCONSOLE_SCREEN_BUFFER_INFO := Buffer_Info_Rec'Access;

-- SUPPORTING SERVICES --
-----

procedure Get_Buffer_Info is
begin
  if GetConsoleScreenBufferInfo (Output_Buffer, Buffer_Info) =
WIN32_ERROR then
    raise Buffer_Info_Error;
  end if;
end Get_Buffer_Info;

-- CURSOR CONTROL --
-----

function Cursor_Visible return Boolean is
  Cursor : aliased CONSOLE_CURSOR_INFO;
begin

```

```

    if GetConsoleCursorInfo (Output_Buffer,
    Cursor'Unchecked_Access) = 0 then
        raise Cursor_Get_Error;
    end if;
    return Cursor.Visible = 1;
end Cursor_Visible;

procedure Set_Cursor (Visible : in Boolean) is
    Cursor : aliased CONSOLE_CURSOR_INFO;
begin
    if GetConsoleCursorInfo (Output_Buffer,
    Cursor'Unchecked_Access) = 0 then
        raise Cursor_Set_Error;
    end if;
    if Visible = True then
        Cursor.Visible := 1;
    else
        Cursor.Visible := 0;
    end if;
    if SetConsoleCursorInfo (Output_Buffer,
    Cursor'Unchecked_Access) = 0 then
        raise Cursor_Set_Error;
    end if;
end Set_Cursor;

function Where_X return X_Pos is
begin
    Get_Buffer_Info;
    return X_Pos (Buffer_Info_Rec.Cursor_Pos.X);
end Where_X;

function Where_Y return Y_Pos is
begin
    Get_Buffer_Info;
    return Y_Pos (Buffer_Info_Rec.Cursor_Pos.Y);
end Where_Y;

procedure Goto_XY
(X : in X_Pos := X_Pos'First;
 Y : in Y_Pos := Y_Pos'First) is
    New_Pos : COORD := (SHORT (X), SHORT (Y));
begin
    Get_Buffer_Info;
    if New_Pos.X > Buffer_Info_Rec.Size.X then
        New_Pos.X := Buffer_Info_Rec.Size.X;
    end if;
    if New_Pos.Y > Buffer_Info_Rec.Size.Y then
        New_Pos.Y := Buffer_Info_Rec.Size.Y;
    end if;
    if SetConsoleCursorPosition (Output_Buffer, New_Pos) =
WIN32_ERROR then
        raise Cursor_Pos_Error;
    end if;
end Goto_XY;

-----  

-- COLOR CONTROL --
-----

function Get_Foreground return Color_Type is
begin
    Get_Buffer_Info;
    return Color_Type_Value (Buffer_Info_Rec.Attrib.Foreground);
end Get_Foreground;

function Get_Background return Color_Type is
begin
    Get_Buffer_Info;
    return Color_Type_Value (Buffer_Info_Rec.Attrib.Background);
end Get_Background;

procedure Set_Foreground (Color : in Color_Type := Gray) is
    Attr : Attribute;
begin
    Get_Buffer_Info;
    Attr.Foreground := Color_Value (Color);
    Attr.Background := Buffer_Info_Rec.Attrib.Background;
    if SetConsoleTextAttribute (Output_Buffer, Attr) =
WIN32_ERROR then
        raise Set_Attribute_Error;
    end if;
end Set_Foreground;

procedure Set_Background (Color : in Color_Type := Black) is
    Attr : Attribute;
begin
    Get_Buffer_Info;
    Attr.Foreground := Buffer_Info_Rec.Attrib.Foreground;
    Attr.Background := Color_Value (Color);
    if SetConsoleTextAttribute (Output_Buffer, Attr) =
WIN32_ERROR then
        raise Set_Attribute_Error;
    end if;
end Set_Background;

-----  

-- SCREEN CONTROL --
-----

procedure Clear_Screen (Color : in Color_Type := Black) is
    Length : DWORD;
    Attr : Attribute;
    Home : COORD := (0, 0);
begin
    Get_Buffer_Info;
    Length := DWORD (Buffer_Info_Rec.Size.X) * DWORD
(Buffer_Info_Rec.Size.Y);
    Attr.Background := Color_Value (Color);
    Attr.Foreground := Buffer_Info_Rec.Attrib.Foreground;
    if SetConsoleTextAttribute (Output_Buffer, Attr) =
WIN32_ERROR then
        raise Set_Attribute_Error;
    end if;
    if FillConsoleOutputAttribute (Output_Buffer, Attr, Length,
Home, NumBytes_Access) = WIN32_ERROR then
        raise Fill_Attribute_Error;
    end if;
    if FillConsoleOutputCharacter (Output_Buffer, ' ', Length,
Home, NumBytes_Access) = WIN32_ERROR then
        raise Fill_Char_Error;
    end if;
    if SetConsoleCursorPosition (Output_Buffer, Home) =
WIN32_ERROR then
        raise Cursor_Position_Error;
    end if;
end Clear_Screen;

-----  

-- SOUND CONTROL --
-----

procedure Bleep is
begin
    if MessageBeep (16#FFFFFF#) = WIN32_ERROR then
        raise Beep_Error;
    end if;
end Bleep;

-----  

-- INPUT CONTROL --
-----

function Get_Key return Character is
    Temp : Integer;
begin
    Temp := GetCh;
    if Temp = 16#00E0# then
        Temp := 0;
    end if;
    return Character'Val (Temp);
end Get_Key;

function Key_Available return Boolean is
begin
    if KBHit = 0 then
        return False;
    else
        return True;
    end if;
end Key_Available;

```

```

    end if;
end Key_Available;

begin
----- WIN32 INITIALIZATION -----
-----

Output_Buffer := GetStdHandle (STD_OUTPUT_HANDLE);
if Output_Buffer = INVALID_HANDLE_VALUE then
  raise Invalid_Handle_Error;
end if;

end NT_Console;

```

Secret (cliquez pour afficher)

Code : Ada - NT_Console.ads

```

-----
-- File: nt_console.ads
-- Description: Win95/NT console support
-- Rev: 0.2
-- Date: 08-june-1999
-- Author: Jerry van Dijk
-- Mail: jdijk@acm.org
-- Copyright (c) Jerry van Dijk, 1997, 1998, 1999
-- Billies Holidaystraat 28
-- 2324 LK LEIDEN
-- THE NETHERLANDS
-- tel int + 31 71 531 43 65
--
-- Permission granted to use for any purpose, provided this
copyright
-- remains attached and unmodified.
--
-- THIS SOFTWARE IS PROVIDED ''AS IS'' AND WITHOUT ANY EXPRESS OR
-- IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
-- WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.
-----

package NT_Console is

  -- TYPE DEFINITIONS --
  subtype X_Pos is Natural range 0 .. 79;
  subtype Y_Pos is Natural range 0 .. 24;

  type Color_Type is
    (Black, Blue, Green, Cyan, Red, Magenta, Brown, Gray,
     Light_Blue, Light_Green, Light_Cyan, Light_Red,
     Light_Magenta, Yellow, White);

  -- EXTENDED PC KEYS --
  Key_Alt_Escape      : constant Character := Character'Val
  (16#01#);
  Key_Control_At     : constant Character := Character'Val
  (16#03#);
  Key_Alt_Backspace  : constant Character := Character'Val
  (16#0E#);
  Key_BackTab        : constant Character := Character'Val
  (16#0F#);
  Key_Alt_Q          : constant Character := Character'Val
  (16#10#);
  Key_Alt_W          : constant Character := Character'Val
  (16#11#);
  Key_Alt_E          : constant Character := Character'Val
  (16#12#);
  Key_Alt_R          : constant Character := Character'Val
  (16#13#);
  Key_Alt_T          : constant Character := Character'Val
  (16#14#);
  Key_Alt_Y          : constant Character := Character'Val
  (16#15#);
  Key_Alt_U          : constant Character := Character'Val
  (16#16#);
  Key_Alt_I          : constant Character := Character'Val
  (16#17#);
  Key_Alt_O          : constant Character := Character'Val
  (16#18#);
  Key_Alt_P          : constant Character := Character'Val
  (16#19#);
  Key_Alt_LBracket   : constant Character := Character'Val
  (16#1A#);
  Key_Alt_RBracket   : constant Character := Character'Val
  (16#1B#);
  Key_Alt_Return     : constant Character := Character'Val
  (16#1C#);
  Key_Alt_A          : constant Character := Character'Val
  (16#1E#);
  Key_Alt_S          : constant Character := Character'Val
  (16#1F#);
  Key_Alt_D          : constant Character := Character'Val
  (16#20#);
  Key_Alt_F          : constant Character := Character'Val
  (16#21#);
  Key_Alt_G          : constant Character := Character'Val
  (16#22#);
  Key_Alt_H          : constant Character := Character'Val
  (16#23#);
  Key_Alt_J          : constant Character := Character'Val
  (16#24#);
  Key_Alt_K          : constant Character := Character'Val
  (16#25#);
  Key_Alt_L          : constant Character := Character'Val
  (16#26#);
  Key_Alt_Semicolon : constant Character := Character'Val
  (16#27#);
  Key_Alt_Quote      : constant Character := Character'Val
  (16#28#);
  Key_Alt_Backquote  : constant Character := Character'Val
  (16#29#);
  Key_Alt_Backslash  : constant Character := Character'Val
  (16#2B#);
  Key_Alt_Z          : constant Character := Character'Val
  (16#2C#);
  Key_Alt_X          : constant Character := Character'Val
  (16#2D#);
  Key_Alt_C          : constant Character := Character'Val
  (16#2E#);
  Key_Alt_V          : constant Character := Character'Val
  (16#2F#);
  Key_Alt_B          : constant Character := Character'Val
  (16#30#);
  Key_Alt_N          : constant Character := Character'Val
  (16#31#);
  Key_Alt_M          : constant Character := Character'Val
  (16#32#);
  Key_Alt_Comma     : constant Character := Character'Val
  (16#33#);
  Key_Alt_Period    : constant Character := Character'Val
  (16#34#);
  Key_Alt_Slash      : constant Character := Character'Val
  (16#35#);
  Key_Alt_KPStar    : constant Character := Character'Val
  (16#37#);
  Key_F1             : constant Character := Character'Val
  (16#3B#);
  Key_F2             : constant Character := Character'Val
  (16#3C#);
  Key_F3             : constant Character := Character'Val

```

```

(16#3D#);
  Key_F4      : constant Character := Character'Val
(16#3E#);
  Key_F5      : constant Character := Character'Val
(16#3F#);
  Key_F6      : constant Character := Character'Val
(16#40#);
  Key_F7      : constant Character := Character'Val
(16#41#);
  Key_F8      : constant Character := Character'Val
(16#42#);
  Key_F9      : constant Character := Character'Val
(16#43#);
  Key_F10     : constant Character := Character'Val
(16#44#);
  Key_Home    : constant Character := Character'Val
(16#47#);
  Key_Up      : constant Character := Character'Val
(16#48#);
  Key_PageUp  : constant Character := Character'Val
(16#49#);
  Key_Alt_KPMinus : constant Character := Character'Val
(16#4A#);
  Key_Left    : constant Character := Character'Val
(16#4B#);
  Key_Center  : constant Character := Character'Val
(16#4C#);
  Key_Right   : constant Character := Character'Val
(16#4D#);
  Key_Alt_KPPlus : constant Character := Character'Val
(16#4E#);
  Key_End     : constant Character := Character'Val
(16#4F#);
  Key_Down    : constant Character := Character'Val
(16#50#);
  Key_PageDown : constant Character := Character'Val
(16#51#);
  Key_Insert   : constant Character := Character'Val
(16#52#);
  Key_Delete   : constant Character := Character'Val
(16#53#);
  Key_Shift_F1 : constant Character := Character'Val
(16#54#);
  Key_Shift_F2 : constant Character := Character'Val
(16#55#);
  Key_Shift_F3 : constant Character := Character'Val
(16#56#);
  Key_Shift_F4 : constant Character := Character'Val
(16#57#);
  Key_Shift_F5 : constant Character := Character'Val
(16#58#);
  Key_Shift_F6 : constant Character := Character'Val
(16#59#);
  Key_Shift_F7 : constant Character := Character'Val
(16#5A#);
  Key_Shift_F8 : constant Character := Character'Val
(16#5B#);
  Key_Shift_F9 : constant Character := Character'Val
(16#5C#);
  Key_Shift_F10 : constant Character := Character'Val
(16#5D#);
  Key_Control_F1 : constant Character := Character'Val
(16#5E#);
  Key_Control_F2 : constant Character := Character'Val
(16#5F#);
  Key_Control_F3 : constant Character := Character'Val
(16#60#);
  Key_Control_F4 : constant Character := Character'Val
(16#61#);
  Key_Control_F5 : constant Character := Character'Val
(16#62#);
  Key_Control_F6 : constant Character := Character'Val
(16#63#);
  Key_Control_F7 : constant Character := Character'Val
(16#64#);
  Key_Control_F8 : constant Character := Character'Val
(16#65#);
  Key_Control_F9 : constant Character := Character'Val
(16#66#);
  Key_Control_F10 : constant Character := Character'Val
(16#67#);
  Key_Control_F1 : constant Character := Character'Val
(16#68#);
  Key_Alt_F2    : constant Character := Character'Val
(16#69#);
  Key_Alt_F3    : constant Character := Character'Val
(16#6A#);
  Key_Alt_F4    : constant Character := Character'Val
(16#6B#);
  Key_Alt_F5    : constant Character := Character'Val
(16#6C#);
  Key_Alt_F6    : constant Character := Character'Val
(16#6D#);
  Key_Alt_F7    : constant Character := Character'Val
(16#6E#);
  Key_Alt_F8    : constant Character := Character'Val
(16#6F#);
  Key_Alt_F9    : constant Character := Character'Val
(16#70#);
  Key_Alt_F10   : constant Character := Character'Val
(16#71#);
  Key_Control_Left : constant Character := Character'Val
(16#73#);
  Key_Control_Right : constant Character := Character'Val
(16#74#);
  Key_Control_End : constant Character := Character'Val
(16#75#);
  Key_Control_PageDown : constant Character := Character'Val
(16#76#);
  Key_Control_Home : constant Character := Character'Val
(16#77#);
  Key_Alt_1     : constant Character := Character'Val
(16#78#);
  Key_Alt_2     : constant Character := Character'Val
(16#79#);
  Key_Alt_3     : constant Character := Character'Val
(16#7A#);
  Key_Alt_4     : constant Character := Character'Val
(16#7B#);
  Key_Alt_5     : constant Character := Character'Val
(16#7C#);
  Key_Alt_6     : constant Character := Character'Val
(16#7D#);
  Key_Alt_7     : constant Character := Character'Val
(16#7E#);
  Key_Alt_8     : constant Character := Character'Val
(16#7F#);
  Key_Alt_9     : constant Character := Character'Val
(16#80#);
  Key_Alt_0     : constant Character := Character'Val
(16#81#);
  Key_Alt_Dash  : constant Character := Character'Val
(16#82#);
  Key_Alt_Equals : constant Character := Character'Val
(16#83#);
  Key_Control_PageUp : constant Character := Character'Val
(16#84#);
  Key_F11      : constant Character := Character'Val
(16#85#);
  Key_F12      : constant Character := Character'Val
(16#86#);
  Key_Shift_F11 : constant Character := Character'Val
(16#87#);
  Key_Shift_F12 : constant Character := Character'Val
(16#88#);
  Key_Control_F11 : constant Character := Character'Val
(16#89#);
  Key_Control_F12 : constant Character := Character'Val
(16#8A#);
  Key_Alt_F11   : constant Character := Character'Val
(16#8B#);
  Key_Alt_F12   : constant Character := Character'Val
(16#8C#);
  Key_Control_Up : constant Character := Character'Val
(16#8D#);
  Key_Control_KPDash : constant Character := Character'Val
(16#8E#);

```

```

Key_Control_Center   : constant Character := Character'Val
(16#SF#);
Key_Control_KPPlus  : constant Character := Character'Val
(16#90#),
Key_Control_Down    : constant Character := Character'Val
(16#91#);
Key_Control_Insert   : constant Character := Character'Val
(16#92#);
Key_Control_Delete   : constant Character := Character'Val
(16#93#);
Key_Control_KPSlash  : constant Character := Character'Val
(16#95#);
Key_Control_KPStar   : constant Character := Character'Val
(16#96#);
Key_Control_EHome    : constant Character := Character'Val
(16#97#);
Key_Alt_EUp          : constant Character := Character'Val
(16#98#);
Key_Alt_EPageUp     : constant Character := Character'Val
(16#99#);
Key_Alt_ELeft        : constant Character := Character'Val
(16#9B#);
Key_Alt_ERight       : constant Character := Character'Val
(16#9D#);
Key_Alt_EEnd         : constant Character := Character'Val
(16#9F#);
Key_Alt_EDown        : constant Character := Character'Val
(16#A0#);
Key_Alt_EPageDown   : constant Character := Character'Val
(16#A1#);
Key_Alt_EInsert      : constant Character := Character'Val
(16#A2#);
Key_Alt_EDelete      : constant Character := Character'Val
(16#A3#);
Key_Alt_KPSSlash     : constant Character := Character'Val
(16#A4#);
Key_Alt_Tab          : constant Character := Character'Val
(16#A5#);
Key_Alt_Enter         : constant Character := Character'Val
(16#A6#);

----- CURSOR CONTROL -----

function Cursor_Visible return Boolean;
procedure Set_Cursor (Visible : in Boolean);

function Where_X return X_Pos;
function Where_Y return Y_Pos;

procedure Goto_XY
(X : in X_Pos := X_Pos'First;
Y : in Y_Pos := Y_Pos'First);

----- COLOR CONTROL -----

function Get_Foreground return Color_Type;
function Get_Background return Color_Type;

procedure Set_Foreground (Color : in Color_Type := Gray);
procedure Set_Background (Color : in Color_Type := Black);

----- SCREEN CONTROL -----

procedure Clear_Screen (Color : in Color_Type := Black);

----- SOUND CONTROL -----

procedure Bleep;

----- INPUT CONTROL -----

function Get_Key return Character;
function Key_Available return Boolean;

end NT_Console;

```

Vous savez désormais comment fonctionnent les packages : copiez chacun de ces textes dans un fichier séparé et enregistrez-les sous les noms NT_Console.adb et NT_Console.ads dans le répertoire de votre projet et n'oubliez pas d'écrire la ligne ci-dessous en en-tête de votre projet :

Code : Ada

With NT_Console ; Use NT_Console ;

Le contenu en détail

Voyons maintenant le contenu de ce package. Pour cela, ouvrez le fichier NT_Console.ads (le code source ne nous intéresse pas, nous n'allons observer que les spécifications). Pour les plus curieux, suivez le guide. Pour ceux qui peuvent s'en sortir seuls (et c'est faisable) lisez ce fichier par vous-mêmes.

[C'est l'histoire de trois types...](#)

Ce package est très bien ficelé, il commence par présenter le package : nom, nom de l'auteur, version, date de création, droits... Je vous conseille de vous en inspirer pour plus de lisibilité dans vos packages. Mais venons-en à ce qui nous intéresse. NT_Console commence par définir trois types et seulement trois :

- **X_Pos** : c'est un Natural entre 0 et 79. Il correspond à la position de votre curseur sur une ligne de votre console : le premier emplacement est le numéro 0, le dernier est le numéro 79. Autrement dit, vous pouvez afficher 80 caractères par ligne.
- **Y_Pos** : c'est un Natural entre 0 et 24. Il correspond au numéro de la ligne à laquelle se situe votre curseur. La première ligne est le numéro 0, la dernière la numéro 24 d'où un total de 25 lignes affichables dans la console. A elles deux, des variables de type X_Pos et Y_Pos vous indiquent où se situe votre curseur à l'écran. Attention, encore une fois, le premier emplacement est le numéro (0, 0)
- **Color_Type** : c'est un type énuméré comptant 15 noms de couleur. C'est ce type qui va nous servir à définir la couleur du texte ou de l'arrière plan. Pour ceux qui auraient du mal avec l'anglais, c'est comme si vous aviez ceci :

Code : Ada

```

type Color_Type is
  (Noir, Bleu, Vert, Cyan, Rouge, Magenta, Marron, Gris,
  Bleu_Claire, Vert_Claire, Cyan_Claire, Rouge_Claire,
  Magenta_Claire, Jaune, Blanc);

```

Puis, vient une longue (très longue) liste de constantes correspondant aux valeurs de certaines touches ou combinaisons de touches du clavier.

[Et pour quelques fonctions de plus](#)

Nous en venons donc à l'essentiel : les procédures et fonctions. Elles sont très bien rangées en cinq catégories : contrôle du curseur, contrôle de la couleur, contrôle de l'écran, contrôle du son et contrôle des entrées (entrées-clavier bien sûr).

[Contrôle du curseur](#)

Pour savoir si le curseur est visible ou non, utilisez la fonction `Cursor_Visible()`. Pour définir si le curseur sera visible ou non utilisez la procédure `Set_Cursor()`. Pour connaître la position du curseur, c'est à dire connaître ses coordonnées `X_Pos` et `Y_Pos`, vous pourrez utiliser les fonctions `Where_X` et `Where_Y` (`Where = où`). Enfin, pour modifier cette position, vous

utiliserez la procédure `Goto_XY()` qui prend en paramètre une variable de type `X_Pos` et une de type `Y_Pos`.

Contrôle de la couleur

Il y a deux éléments dont nous pouvons modifier la couleur à l'affichage : la couleur du texte (avant-plan ou premier plan) et la couleur du fond, du surlignage (arrière-plan). C'est à cela que correspondent `Foreground` (avant-plan) et `Background` (arrière-plan).

Comme toujours, deux actions sont possibles : lire et écrire. Si vous souhaitez connaître la couleur de l'arrière plan, vous devez la lire et vous utiliserez donc la fonction `Get_Background` qui vous retournera la couleur de fond. Si vous souhaitez modifier cette couleur, alors vous utiliserez la procédure `Set_Background(une_couleur)`. Même chose pour l'avant-plan bien entendu.

Retenez donc ceci :

- `Foreground` = Avant-plan / `Background` = Arrière-plan
- `Get` = Saisir / `Set` = Définir

Contrôle de l'écran

Cette section, comme la suivante, ne comporte qu'une seule procédure : `Clear_Screen()`. Cette procédure prend en paramètre une couleur (une variable de type `Color_Type`) et, comme son nom l'indique, nettoie l'écran. Plus rien n'est affiché et en plus vous pouvez changer en même temps la couleur de fond de la console. Attention, la couleur de fond de la console est différente de la couleur d'arrière-plan de votre texte !

Contrôle du son

Une seule procédure sans grand intérêt : `Bleep`. Cette procédure se contente d'émettre un bip, comme si Windows avait rencontré une erreur.

Contrôle des entrées clavier

Lorsque l'utilisateur appuie sur les touches du clavier (même sans que votre programme ne l'y ait invité), celui-ci transmet une information à votre ordinateur qui la stocke en mémoire (on parle de mémorie tampon ou de Buffer, souvenez-vous, nous en avions parlé quand nous avions vu l'instruction `Skip_line`). Ainsi, la mémoire tampon peut contenir toute une série de caractères avant même que le programme n'en ait besoin ou bien être vide alors que le programme attend une réponse. La fonction `key_available` vous permet de savoir si une touche a été stockée en mémoire tampon. Les fonctions et procédures `get`, `get_line`, `get_immediate`... se contentent ainsi de pointer dans la mémoire tampon, au besoin en attendant qu'elle se remplit.

La fonction `get_immediate` a elle agi tout à la manière de la procédure `get_immediate` : elle pointe immédiatement dans la mémoire tampon sans attendre que l'utilisateur valide la saisie avec Entrée. Quel intérêt ? Eh bien il y a une petite différence, notamment dans la gestion des touches «spéciales». Par «touches spéciales», j'entends les touches flèches ou F10 par exemple. Essayez ce code :

Code : Ada

```
with nt.console ;           use nt.console ;
with ada.Text_IO ;          use ada.Text_IO ;
with ada.Integer_Text_IO ;   use ada.Integer_Text_IO ;

procedure test is
  c : character ;
begin
  loop
    c := get_key ;
    put("valeur correspondante :") ;
    put(character'pos(c)) ;
    new_line ;
  end loop ;
end test01 ;
```

Vous vous rendrez compte que toute touche spéciale envoie en fait deux caractères : le caractère n°0 suivi d'un second. Ainsi, la flèche gauche envoie le caractère n°0 suivi du caractère n°75. À l'aide de ce petit programme, vous pourrez récupérer les numéros des touches fléchées qui serviront au joueur à diriger le serpent.

La fonction `key_available` quant à elle indique si le buffer est vide ou pas. Si elle renvoie `TRUE`, c'est qu'il y a au moins un caractère en mémoire tampon et que l'on peut le saisir. Chose importante, elle n'attend pas que l'utilisateur appuie sur une touche ! Ainsi, à l'aide de cette fonction, nous pourrons faire en sorte que le serpent continue à avancer sans attendre que l'utilisateur appuie sur une touche ! Nous n'utiliserons `get_key` que dans le cas où `key_available` aura préalablement renvoyé `TRUE`.

... et encore un autre !

Pour mesurer le temps et définir ainsi la vitesse de jeu, nous aurons besoin d'un second package (officiel celui-là) : `Ada.Calendar` !

Le package Ada.calendar est fait pour nous permettre de manipuler l'horloge. Si vous cherchez le fichier correspondant, il porte le nom `a-calend.ads`. Deux types essentiels le composent : le type `Duration` et le type `Time` (même si `Time` est une copie de `Duration`). `Duration` mesure les durées, `Time` correspond aux dates. Ce sont tous deux des nombres décimaux, puisque le temps est mesuré en secondes. Attention toutefois aux problèmes de typage ! Nous allons définir deux variables : temps et durée.

Code : Ada

```
temps : time ;
duree : Duration ;
```

Pour saisir la date actuelle (jour, mois, année, seconde), il faut utiliser la fonction `clock` :

Code : Ada

```
temps := clock ;
```

Il est ensuite possible d'effectuer des opérations, par exemple :

Code : Ada

```
duree := clock - temps ;
```

En soustrayant le temps enregistré dans la variable `temps` au temps actuel fourni par la fonction `clock`, on obtient une durée. Il est également possible de soustraire (ou d'ajouter) une durée à un temps, ce qui donne alors un temps. Il est également possible de comparer des variables de type `Time` ou `Duration`. Enfin, pour afficher une variable de type `Time` ou `Duration`, pensez à la convertir préalablement en `Float`. Pour en savoir plus sur le package Ada.Calendar, n'hésitez pas à éplucher les fichiers `a-calend.adb` et `a-calend.ads`, ils ne sont pas très longs.

Quelques indications

Jouer en temps réel



Nous allons devoir combiner les deux packages dont je viens de vous parler pour parvenir à ce petit exploit. Le serpent avance case par case, chaque avancée prend un certain temps (pour ma part, j'ai pris une durée de 0,2 secondes) durant lequel le joueur peut appuyer sur les touches de son clavier pour modifier la direction du serpent. Une fois ce temps écoulé, le serpent avance d'une case. Cela nous amène à l'algorithme suivant :

Code : Français

```
TANT QUE duree<0.2 REPETER
  | attendre_une_reaction_du_joueur ;
  FIN REPETER
  Avancer_le_serpent ;
```



C'est là qu'intervient le package NT_Console ! Nous ne devons pas attendre que le joueur ait saisi une touche mais simplement

nous demander s'il a appuyé sur une touche. Si oui, on saisit effectivement cette touche, sinon on continue à parcourir notre boucle. Cela nous amène à l'algorithme suivant :

Code : Ada

```
if Key_available
  then c:=get_key ;
end if ;
```

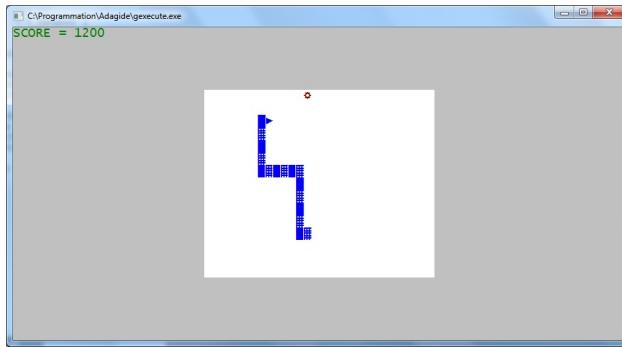
Ainsi, le programme ne reste pas bloqué sur get_key indéfiniment. Cependant, nous attendons que le joueur appuie sur une touche fléchée, pas sur une lettre ! Comme je vous l'ai dit, les touches spéciales envoient deux caractères consécutivement : le numéro 0 puis le véritable caractère. Donc nous devons reprendre le code précédent ainsi :

Code : Ada

```
if Key_available
  then if character'pos(get_key)=0
        then c:=get_key ;
        traitement ;
        end if ;
  end if ;
```

Comment afficher un serpent et une zone de jeu en couleur ?

Pour cela, regardons la capture d'écran que je vous ai fournie, elle n'est pas parfaite mais elle va nous guider :



Jeu du serpent

Que remarque-t-on ? Tout d'abord, le fond de la console est gris, contrairement à faire de jeu qui elle est blanche et au serpent qui est dessiné en bleu par-dessus faire de jeu. Nous avons donc trois niveau de couleur :

- La couleur de fond de la console : elle sera obtenue lorsque vous utiliser la procédure `clear_screen()`.
- La couleur de la zone de jeu : elle est obtenue en fixant la couleur d'arrière-plan et en affichant de simples espaces. Attention à ne pas afficher de caractères avec un arrière-plan blanc en dehors de cette zone. Notamment si vous affichez du texte comme le score ou un message de défaite.
- La couleur des items affichés sur la zone de jeu : ces items comme les morceaux du serpent ou l'objet qu'il doit manger, doivent avoir un arrière-plan blanc et un avant-plan bleu (ou n'importe quelle autre couleur de votre choix).

Pensez également que lorsque votre serpent avancera d'une case, il ne suffira pas d'afficher le nouveau serpent, il faudra également penser à effacer l'ancien de l'écran !



Mais d'où tu sorts tes ronds, tes rectangles et tes triangles ?

C'est simple. Il suffit de réaliser un petit programme qui affiche tous les caractères du numéro 0 au numéro 255 (à l'aide des attributs `'pos` et `'val`). On découvre ainsi plein de caractères bizarroïdes comme le trèfle, le cœur, le pique et le carreau ou des flèches, des canards... Pour les rectangles du corps du serpent, j'ai utilisé `character'val(219)` et `character'val(178)`, pour les triangles j'ai utilisé `character'val(16)`, `character'val(17)`, `character'val(30)` ou `character'val(51)` et quant à l'espèce de soleil, il s'agit de `character'val(15)`. Mais ne vous contentez pas de prendre mot à mot mes propositions, faites des essais et trouvez les symboles qui vous paraissent les plus appropriés.

Par où commencer ?

Nous en sommes au troisième TP, il serait bon que vous commenciez à établir vous-même votre approche du problème. Je ne vais donc pas détailler cette partie. Voici comment j'aborderais ce TP (en entrecoupant chaque étape de divers tests pour trouver d'éventuels bogues) :

- Création des types nécessaires pour faire un serpent.
- Réalisation des procédures d'affichage de l'aire de jeu et du serpent.
- Réalisation des procédures de déplacement et d'agrandissement du serpent + actualisation éventuelle des procédures d'affichage.
- Réalisation des procédures permettant au Joueur de piloter le Serpent au clavier.
- Mise en place des règles : interdiction de sortir de l'espace de jeu, de mordre le corps du serpent, d'effectuer des déviations.
- Réalisation des procédures générant l'item à avaler (appelé Anneau dans mon code) et actualisation des règles : «si le serpent mange l'anneau alors il grandit», «un anneau ne peut apparaître directement sur le serpent».
- Éventuels débogages et ajouts de fonctionnalités.

Voilà pour le plan de bataille. Il est succinct mais devrait vous fournir des objectifs partiels facilement atteignables. Je me répète mais n'hésitez pas à effectuer des tests réguliers et approfondis pour être sûr de ne pas avoir créé de bogues. Par exemple, ce n'est pas parce que votre serpent avance correctement vers le haut, qu'il avancera correctement vers la droite ou après avoir effectué un ou deux virages. Et en cas de difficultés persistantes, n'hésitez pas à poser des questions.

Une solution possible

Le fichier principal :

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
with nt_console ;
with Snake_Variables ;
with Snake_Programs ;
with Snake_Screen ;
```

```
procedure Snake is
  Serpent      : T_Serpent ;
begin
  Set_Cursor(false) ;
  print_ecran(snake) ;
  print_ecran(ready) ;
  print_ecran(start) ;
  Clear_screen(Couleur_Ecran) ;
  print_plateau ;
  Init_Serpent(Serpent) ;
  Print_Serpent(Serpent) ;
  Game(Serpent) ;
end Snake ;
```

Le package contenant quelques types et surtout les variables nécessaires au programme :

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
with nt_console ;           use nt_console ;

package Snake_Variables is
    type T_Coord is record
        x,y : integer := 0 ;
    end record;

    Longueur      : Natural          := 30 ;
    Hauteur       : Natural          := 15 ;

    subtype Intervalle_Alea is natural range 1..Longueur*Hauteur ;

    HDecalage   : Natural          := (X_Pos'last - Pos'first + 1 - Longueur)/2 ;
    VDecalage   : Natural          := (Y_Pos'last - Y_Pos'first + 1 - Hauteur)/2 ;

    Couleur_Ecran : Color_Type     := gray ;
    Couleur_Fond  : Color_Type     := white ;
    Couleur_Texte : Color_Type     := light_blue ;
    Couleur_Anneau: Color_Type     := red ;

    Duree        : Duration        := 0.2 ;
    Score        : Natural          := 0 ;

    touche       : Character ;
end Snake_Variables ;
```

Le package contenant la plupart des programmes servant au jeu :

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
with nt_console ;           use nt_console ;
with ada.text_io ;          use ada.Text_IO ;
with ada.Integer_Text_IO ;
ada.Integer_Text_IO ;        use ada.Calendar ;

package body Snake_Programs is
    -- AFFICHAGE PLATEAU --
    -- CREATION SERPENT --
    procedure print_plateau is
    begin
        print_score ;
        new_line(Count(VDecalage)) ;
        for j in 1..Hauteur loop
            set_background(Couleur_Ecran) ;
            for i in 1..HDecalage loop
                put(' ') ;
            end loop ;
            set_background(Couleur_Fond) ;
            for i in 1..Longueur loop
                put(' ') ;
            end loop ;
            new_line ;
        end loop ;
    end print_plateau ;

    procedure print_score is
    begin
        set_foreground(green) ;
        set_background(Couleur_Ecran) ;
        goto_XY(0,0) ;
        put("SCORE") ; put(Score,4) ;
        set_foreground(Couleur_Texte) ;
        set_background(Couleur_Fond) ;
    end print_score ;
    procedure Init_Serpent(Serpent : in out T_Serpent) is
    begin
        Append(Serpent.corps,(15,7)) ;           --placement des
        anneaux du serpent
        Append(Serpent.corps,(15,8)) ;
        Append(Serpent.corps,(15,9)) ;
        Append(Serpent.corps,(14,9)) ;
        Append(Serpent.corps,(13,9)) ;
        Serpent.curseur := First(Serpent.corps) ; --placement du
        curseur sur la tête du serpent
        Serpent.direction := (0,-1) ;             --direction vers
        le haut
    end Init_Serpent ;

    procedure move(Serpent : in out T_Serpent) is
    begin
        coord: T_coord ;
        begin
            efface_queue(Serpent) ;           --on
            efface la queue
            coord.x := Element(Serpent.curseur).x +
            Serpent.direction.x ;
            coord.y := Element(Serpent.curseur).y +
            Serpent.direction.y ;           --on
            Prepend(Serpent.corps,coord) ;
            ajoute une nouvelle tête
            Serpent.curseur := Last(Serpent.corps) ;
            Delete(Serpent.corps,Serpent.curseur) ; --on
            supprime la queue
            Serpent.curseur := First(Serpent.corps) ;
            print_serpent(serpent) ;           --on
            affiche le nouveau corps
        end move ;
    end move ;

    procedure grow(Serpent : in out T_Serpent) is
    begin
        coord: T_coord ;
        begin
            coord.x := First_Element(Serpent.corps).x +
            Serpent.direction.x ;
            coord.y := First_Element(Serpent.corps).y +
            Serpent.direction.y ;           --on
            Prepend(Serpent.corps,coord) ;
            ajoute une nouvelle tête
            Serpent.curseur := First(Serpent.corps) ;
            print_serpent(serpent) ;           --on
            affiche le nouveau corps
        end grow ;
    end grow ;

    function est_sorti(Serpent : T_Serpent) return boolean is
    tete : T_Coord ;
    begin
        tete := First_Element(Serpent.corps) ;
        if tete.x < 1 or tete.x > Longueur or tete.y < 1 or tete.y >
        hauteur
            then return true ;
            else return false ;
        end if ;
    end est_sorti ;

    function est_mordu(Serpent : T_Serpent) return boolean is
    tete : T_Coord ;
    Serpent2 : T_Serpent := Serpent ;
    begin
        tete := First_Element(Serpent2.corps) ;
```

```

Delete first(Serpent2.corps) ;
  if Find(Serpent2.corps,tete) /= No_Element
    then return true ;
    else return false ;
  end if ;
end est_mordu ;

function a_mange(Serpent : T_Serpent ; Anneau : T_Coord) return
boolean is
  tete : T_Coord ;
begin
  tete := First_Element(Serpent.corps) ;
  if tete = Anneau
    then return true ;
    else return false ;
  end if ;
end a_mange ;
-----+
-- AFFICHAGE SERPENT --
-----

procedure print_tete(Serpent : in T_Serpent) is
begin
  if serpent.direction.x < 0
    then put(character'val(17)) ; --regard vers la gauche
  elsif serpent.direction.x > 0
    then put(character'val(16)) ; --regard vers la droite
  elsif serpent.direction.y < 0
    then put(character'val(30)) ; --regard vers le haut
    else put(character'val(31)) ; --regard vers le bas
  end if ;
end print_tete ;

procedure print_corps(nb : natural) is
begin
  if nb mod 2 = 0
    then put(character'val(219)) ;
    else put(character'val(178)) ;
  end if ;
end print_corps ;

procedure print_serpent(Serpent : in out T_Serpent) is
begin
  Set_Foreground(Couleur_Texte) ;
  for i in 1..length(Serpent.corps) loop
    Goto_XY(Element(serpent.curseur).x + HDecalage-1,
            Element(serpent.curseur).y + VDecalage-1) ;
    if i = 1
      then print_tete(serpent) ;
      else print_corps(integer(i)) ;
    end if ;
    Next(Serpent.curseur) ;
  end loop ;
  Serpent.curseur := First(Serpent.corps) ;
end print_serpent ;

procedure efface_queue(Serpent : in out T_Serpent) is
begin
  Serpent.curseur := Last(Serpent.corps) ;
  Goto_XY(Element(serpent.curseur).x + HDecalage-1,
          Element(serpent.curseur).y + VDecalage-1) ;
  put(' ') ;
  Serpent.curseur := First(Serpent.corps) ;
end efface_queue ;
-----+
-- GESTION DES ANNEAUX --
-----

function generer(germe : generator ; Serpent : in T_Serpent)
return T_Coord IS
  temp1,temp2 : Natural ;
  anneau : T_Coord ;
BEGIN
  loop
    temp1 := random(germe) ; temp2 := random(germe) ;
    Anneau := (temp1 mod longueur + 1, temp2 mod hauteur +1)
    if Find(Serpent.corps,Anneau) = No_Element
      then return Anneau ;
    end if ;
  end loop ;
end generer ;

procedure print_anneau(anneau : T_Coord) is
begin
  set_foreground(Couleur_Anneau) ;
  Goto_XY(anneau.x+HDecalage-1,anneau.y+VDecalage-1) ;
  put(character'val(15)) ;
  set_foreground(Couleur_Texte) ;
end print_anneau ;
-----+
-- PROGRAMME DE JEU --
-----

procedure Erreur(Message : in String) is
begin
  Goto_XY(HDecalage+5,VDecalage+Hauteur+2) ;
  set_background(Couleur_Ecran) ;
  set_foreground(light_Red) ;
  Put(Message) ; delay 1.0 ;
end Erreur ;

procedure un_tour(Serpent : in out T_Serpent) is
  choix_effectue : boolean := false ;
  Temps : constant Time := clock ;
  New_direction : T_Coord ;
begin
  while clock-Temps<durée loop
    if not choix_effectue and then key_available and then
character'pos(get_key) = 0
      then touche := get_key ;
      case character'pos(touche) is
        when 72 => New_direction := (0,-1) ; --haut
        when 75 => New_direction := (-1,0) ; --gauche
        when 77 => New_direction := (1,0) ; --droite
        when 80 => New_direction := (0,1) ; --bas
        when others => null ;
      end case ;
      if New_direction.x /= Serpent.direction.x and
New_direction.y /= Serpent.direction.y
        then Serpent.direction := New_direction ;
        choix_effectue := true ;
      end if ;
    end if ;
  end loop ;
end un_tour ;

procedure game(Serpent : in out T_Serpent) is
  germe : generator ;
  Anneau : T_Coord ;
begin
  reset(germe) ;
  Anneau := generer(germe,Serpent) ;
  Print_anneau(Anneau) ;
loop
  un_tour(Serpent) ;
  --test pour savoir si le serpent grandit ou avance
  if a_mange(serpent,anneau)
    then grow(serpent) ;
    Score := Score + 100 ;
    Print_score ;
    Anneau := generer(germe,serpent) ;
    Print_anneau(Anneau) ;
  else move(serpent) ;
  end if ;
  --test pour savoir si le serpent meurt
  if est_sorti(serpent)
    then Erreur("Vous " & character'val(136) & "tes sortis
!") ; exit ;
  elsif est_mordu(serpent)
    then Erreur("Vous vous " & character'val(136) & "tes
mordu !") ; exit ;
  end if ;

```

```

        end loop ;
    end game ;

end Snake_Programs ;

```

Code : Ada

```

with Snake_Variables ;           use Snake_Variables ;
with ada.Numerics.Discrete_Random ; use ada.Containers ;
with ada.Containers.Doubly_Linked_Lists ; use ada.Containers ;

package Snake_Programs is

    package P_Liste is
        new Ada.Containers.Doubly_Linked_Lists(T_coord) ;
        use P_Liste ;
    end P_Liste ;

    package P_Aleatoire is
        new ada.Numerics.Discrete_Random(Intervalle_Alea) ;
        use P_Aleatoire ;
    end P_Aleatoire ;

    Type T_Serpent is record
        corps      : List ;
        Curseur    : Cursor ;
        direction   : T_Coord ;
    end record ;

    procedure print_plateau ;
    procedure print_score ;

    procedure Init_Serpent(Serpent : in out T_Serpent) ;
    procedure move(Serpent : in out T_Serpent) ;
    procedure grow(Serpent : in out T_Serpent) ;
    function est_sorti(Serpent : T_Serpent) return boolean ;
    function est_mordu(Serpent : T_Serpent) return boolean ;

    procedure print_tete(Serpent : in T_Serpent) ;
    procedure print_corps(natural) ;
    procedure print_serpent(Serpent : in out T_Serpent) ;
    procedure efface_queue(Serpent : in out T_Serpent) ;

    function generer(germe : generator ; Serpent : T_Serpent)
        return T_Coord ;
    procedure print_anneau(anneau : T_Coord) ;

    procedure Erreur(Message : in String) ;
    procedure un_tour(Serpent : in out T_Serpent) ;
    procedure game(Serpent : in out T_Serpent) ;

end Snake_Programs ;

```

Enfin, le package servant à l'affichage de mes écrans de titre :

[Secret](#) (cliquez pour afficher)

Code : Ada

```

with ada.Text_IO ;           use ada.Text_IO ;
with NT_Console ;           use NT_Console ;
With Snake_Variables ;       use Snake_Variables ;

package body Snake_Screen is

    procedure print_line(line : string) is
    begin
        for i in line'range loop
            case line(i) is
                when 'R' => set_background(Red) ;
                    put(' ') ;
                    set_background(Couleur_Ecran) ;
                when 'G' => set_background(Green) ;
                    put(' ') ;
                    set_background(Couleur_Ecran) ;
                when 'Y' => set_background(Yellow) ;
                    put(' ') ;
                    set_background(Couleur_Ecran) ;
                when '#' => set_background(Black) ;
                    put(' ') ;
                    set_background(Couleur_Ecran) ;
                when others => put(' ') ;
            end case ;
        end loop ;
        new_line ;
    end print_line ;

    procedure print_fichier(name : string) is
        F : file_type ;
    begin
        open(F, In_File, name) ;
        clear_screen(couleur_ecran) ;
        set_background(Couleur_Ecran) ;
        while not end_of_file(f) and not end_of_page(f) loop
            print_line(get_line(f)) ;
        end loop ;
        close(f) ;
    end print_fichier ;

    procedure print_ecran(Ecran : T_Ecran) is
    begin
        case Ecran is
            when SNAKE => print_fichier("Snake.pic") ; delay 2.5 ;
            when START => print_fichier("Start.pic") ; delay 1.0 ;
            when READY => print_fichier("Ready.pic") ; delay 1.5 ;
        end case ;
    end print_ecran ;

```

end Snake_Screen ;

Code : Ada

```

package Snake_Screen is

    type T_Ecran is (START,READY,SNAKE) ;
    procedure print_line(line : string) ;
    procedure print_fichier(name : string) ;
    procedure print_ecran(Ecran : T_Ecran) ;

end Snake_Screen ;

```

Pour finir, je vous transmets également les fichiers .pic (en fait des fichiers textes) servant de support à ces fameux écrans titres (je suis preneur pour tout écran titre pouvant remplacer celui qui ressemble vaguement à un serpent 😊):

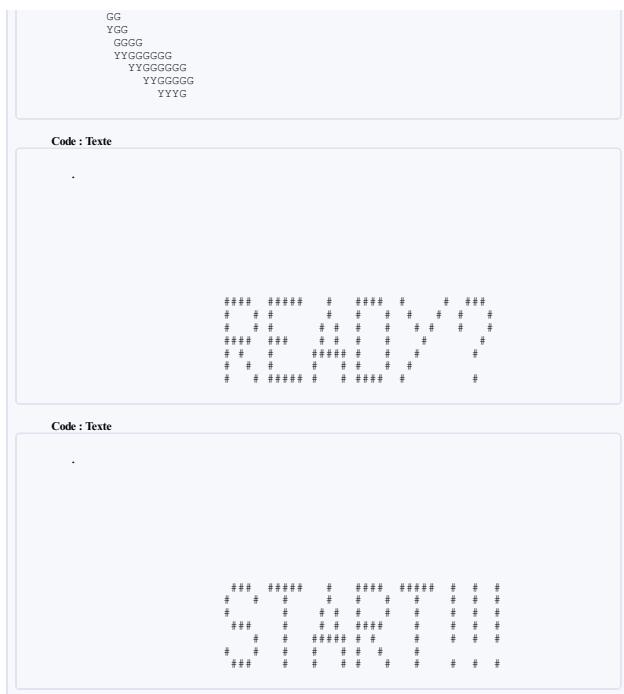
[Secret](#) (cliquez pour afficher)

Code : Texte

```

.
.
.
GGGGGGGG
Y          YG
G          YG
Y          YG
          YG
          GG
          GY
          GGY
          GGGGGGGGGGY
          YGGYGYGYGYGYG
          GGGGY
          GGY
          GGY
          GG

```



Pistes d'amélioration :

- Proposer différents niveaux de difficulté : en faisant varier la vitesse, la taille de l'aire de jeu...
- Ajouter des adversaires ou des obstacles.
- Proposer l'enregistrement des meilleurs scores.
- Proposer des bonus comme des anneaux comptant le double de points ou permettant de réduire votre taille.

Nous avons abordé dans ce chapitre la plupart des notions qui nous manquaient en découvrant des types plus complexes comme les tableaux, les pointeurs, les types abstraits de données... Vous avez donc déjà fait un large tour des possibilités et savez désormais programmer en Ada ! Enfin, en Ada83. Mais la science de l'informatique évolue sans cesse et la programmation n'échappe pas à cette règle. La norme Ada95 a fait entrer le langage Ada dans la catégorie des langages orientés objets, renforcé par la norme Ada2005. Découvrir cette fameuse POO (Programmation Orientée Objet) sera l'objectif principal de la partie IV.



Et quand est-ce que l'on fera des programmes en Ada plus jolis que ces vilaines consoles ? ☺

Ce sera le rôle de la cinquième partie que d'aborder ces notions au travers de GTK, une bibliothèque qui nous permettra de créer de jolies fenêtres avec de jolis boutons et tout et tout ! ☺ Les plus curieux (ou pressés) pourront s'y rendre directement, car il vous sera alors et déjà possible d'obtenir des résultats. Toutefois, pour des applications plus poussées et pour une meilleure compréhension, il sera nécessaire de lire la partie IV, la bibliothèque Gtk étant construite sur le mode « *orienté objet* ».

Partie 4 : Ada : Notions avancées et Programmation Orientée Objet

Nous avons abordé dans la partie précédente des notions essentielles de programmation, mais nous n'avons pas encore découvert la Programmation Orientée Objet. Cette partie n'a pas pour but de révolutionner votre façon de coder mais plutôt d'y apporter des améliorations. Créer des packages plus génériques, gérer des erreurs qui pourraient survenir, gérer de manière plus dynamique vos allocations/désallocations de mémoire...

Comme précédemment, elle comportera un fil conducteur autour duquel viendront se greffer des chapitres un peu plus hors sujet (ou plutôt hors thème). Mon fil conducteur ne sera plus les tableaux ou les pointeurs, mais les packages et la Programmation Orientée Objet.



Je le sais bien, mais il nous reste beaucoup à dire sur les packages. Nous allons aborder ce que l'on appelle les **classes** et les **méthodes**. Que les néophytes se rassurent, tout ira bien et nous prendrons plus de temps qu'il n'en faut pour comprendre ce que cela signifie. Et que les aficionados de la programmation orientée objet calment leurs ardeurs : Ada gère les notions de classes d'une manière particulière, très différente des C++, Java et autres Python.

Algorithmique : tri et complexité

Lors du chapitre sur les tableaux (partie III), nous avions abordé l'algorithme de tri par sélection (Exercice 3). Cet algorithme consistait à trier un tableau d'entiers du plus petit au plus grand en comparant progressivement les éléments d'un tableau avec leurs successeurs, quitte à effectuer un échange (n'hésitez pas à relire cet exercice si besoin). Comme je vous l'avais alors dit, cet algorithme de tri est très rudimentaire et sa **complexité** était forte.



Ce sont les deux questions auxquelles nous allons tenter de répondre. Nous commencerons par évoquer quelques algorithmes de tri (sans chercher à être exhaustif) : tri par sélection (appelé), tri à bulles, tri par insertion, tri rapide, tri fusion et tri par tas. Je vous expliquerai le principe avant de vous proposer un implémentations possible. Ensuite, nous expliquerons en quoi consiste la complexité d'un algorithme et aborderons les aspects mathématiques nécessaires de ce chapitre. Enfin, nous étudierons la complexité de ces différents algorithmes de manière empirique en effectuant quelques mesures.

Ce chapitre est davantage un chapitre d'algorithmique que de programmation. Aucune nouvelle notion de programmation ne sera abordée. En revanche, nous étudierons différentes techniques de tri, parfois surprenantes. Qui plus est, l'étude de complexité, si elle se veut empirique, fera toutefois appel à des notions mathématiques peut-être compliquées pour certains. Pour toutes ces raisons, ce chapitre n'est pas obligatoire pour la poursuite du cours.

Algorithmes de tri lents

Commençons par évoquer les algorithmes les plus lents (cette notion de lenteur sera précisée et nuancée plus tard) : tri par sélection, tri par insertion et tri à bulles. Nous aurons par la suite besoin de quelques programmes essentiels : initialisation et affichage d'un tableau, échange de valeurs. Je vous fournis de nouveau le code nécessaire ci-dessous.

Secret (cliquez pour afficher)

Code : Ada

```
function init return T_Tableau is
    type Tableau;
    subtype Intervalle is integer range 1..100;
    package Aleatoire is
        new Ada.numerics.discrete_random(Intervalle);
    end;
    use Aleatoire;
    Hasard : generator;
begin
    Reset(Hasard);
    for i in T'range loop
        T(i) := random(Hasard);
    end loop;
    return T;
end init;

--Afficher affiche les valeurs d'un tableau sur une même ligne
procedure Afficher(T : T_Tableau) is
begin
    for i in T'range loop
        Put(T(i),4);
    end loop;
    end Afficher;

--Echanger est une procédure qui échange deux valeurs : a vaudra b et b vaudra a
procedure Echanger(a : in out integer; b : in out integer) is
    c : integer;
begin
    c := a;
    a := b;
    b := c;
end Echanger;
```

Tri par sélection (ou par extraction)

Principe

1	2	3	4	5
7	3	18	13	7

Nous allons trier le tableau ci-dessus par sélection. Le principe est simple : nous allons chercher le minimum du tableau et l'échanger avec le premier élément :

Rang 1				
1	2	3	4	5
7	3	18	13	7

Rang 2				
1	2	3	4	5
3	7	18	13	7

Puis on réitere l'opération : on recherche le minimum du tableau (privé toutefois du premier élément) et on l'échange avec le second (ici pas d'échange nécessaire).

Rang 2				
1	2	3	4	5
3	7	18	13	7

On réitere à nouveau l'opération : on recherche le minimum du tableau (privé cette fois des deux premiers éléments) et on l'échange avec le troisième.

Rang 3				
1	2	3	4	5
3	7	18	13	7

Rang 3				
1	2	3	4	5
3	7	18	13	7

On réitere ensuite avec le quatrième élément et on obtient ainsi un tableau entièrement ordonné par ordre croissant. C'est l'un des algorithmes les plus simples et intuitifs. Pour aller plus loin ou pour davantage d'explications, vous pouvez consulter le [tutoriel de K-Phoen](#) à ce sujet (langage utilisé : C).

Mise en œuvre

Je vous fournis à nouveau le code source de l'algorithme de tri par sélection quelque peu modifié par rapport à la fois précédente :

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
function RangMin(T : T_Tableau ; debut : integer ; fin : integer)
return integer is
  Rang : integer := debut ;
  Min : integer := T(debut) ;
begin
  for i in debut..fin loop
    if T(i)<Min
      then Min := T(i) ;
      Rang := i ;
    end if ;
  end loop ;
  return Rang ;
end RangMin ;

function Trier(Tab : T_Tableau) return T_Tableau is
  T : T_Tableau := Tab ;
begin
  for i in T'range loop
    Echanger(T(i),T(RangMin(T,i,T'last))) ;
  end loop ;
  return T ;
end Trier ;
```

Tri par insertion

[Principe](#)

Le principe du tri par insertion est le principe utilisé par tout un chacun pour ranger ses cartes : on les classe au fur et à mesure. Imaginons que dans une main de 5 cartes, nous avons déjà trié les trois premières, nous arrivons donc à la quatrième carte. Que fait-on ? Eh bien c'est simple, on la sort du paquet de cartes pour l'insérer à la bonne place dans la partie des cartes déjà triées. Exemple :

1	2	3	4	5
5	9	10	6	7

1	2	3	4	5
5	6	9	10	7

Il faudra ensuite s'occuper du cinquième élément mais là n'est pas la difficulté, car vous vous en doutez peut-être, mais il est impossible d'insérer un élément entre deux autres dans un tableau. Il n'y a pas de case n°1,5 ! Donc «réinsérez» le quatrième élément dans le tableau consistera en fait à décaler tous ceux qui sont supérieurs d'un cran vers la droite. Pour aller plus loin ou pour davantage d'explications, vous pouvez consulter le [tutoriel de bluestorm](#) à ce sujet (langage utilisé : C) et sa suite utilisant les listes (langage utilisé : OCaml).

[Mise en œuvre](#)

Je vous invite à rédiger cet algorithme par vous-même avant de consulter la solution que je vous propose, c'est la meilleure façon de savoir si vous avez compris le principe ou non (d'autant plus que cet algorithme est relativement simple).

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
function tri_insertion(T : T_Tableau) return T_Tableau is
  Tab : T_Tableau := T ;
begin
  for i in T'first + 1 .. T'last loop
    for j in reverse T'first + 1..i loop
      exit when Tab(j-1)<=Tab(j) ;
      echanger(Tab(j-1),Tab(j)) ;
    end loop ;
  end loop ;
  return Tab ;
end Tri_insertion ;
```

L'instruction `EXIT` située au cœur des deux boucles peut être remplacée par un `IF` :

Code : Ada

```
if Tab(j-1)<=Tab(j)
  then exit ;
  else echanger(Tab(j-1),Tab(j)) ;
end if ;
```

Tri à bulles (ou par propagation)

[Principe](#)

Voici enfin le plus «mauvais» algorithme. Comme les précédents il est plutôt intuitif mais n'est vraiment pas efficace. L'algorithme de tri par propagation consiste à inverser deux éléments successifs s'ils ne sont pas classés dans le bon ordre et à recommencer jusqu'à ce que l'on parcourt le tableau sans effectuer une seule inversion. Ainsi, un petit élément sera échangé plusieurs fois avec ses prédecesseurs jusqu'à atteindre son emplacement définitif : il va se propager peu à peu, ou «monter» vers la bonne position lentement, comme une bulle d'air remonte peu à peu à la surface (j'ai fâné d'un poète aujourd'hui 😊).

Je ne pense pas qu'un exemple soit nécessaire. L'idée (simpissime) est donc de parcourir plusieurs fois le tableau jusqu'à obtenir la solution. Vous vous en doutez, cette méthode est assez peu évolutée et demandera de nombreux passages pour obtenir un tableau rangé. Pour aller plus loin ou pour davantage d'explications, vous pouvez consulter le [tutoriel de shareman](#) à ce sujet (langage utilisé : C++).

[Mise en œuvre](#)

Là encore, vous devriez être capable en prenant un peu de temps, de coder cet algorithme par vous-même.

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
function Tri_Bulles(T : T_Tableau) return T_Tableau is
  Tab : T_Tableau := T ;
  Echange_effectue : boolean := true ;
begin
  while Echange_effectue loop
    --possibilité d'afficher le tableau en écrivant ici
    --Afficher(Tab) ;
    Echange_effectue := false ;
    for i in Tab'first .. Tab'last-1 loop
      if Tab(i) > Tab(i+1)
        then echanger(Tab(i), Tab(i+1)) ;
        Echange_effectue := true ;
      end if ;
    end loop ;
  end loop ;
  return Tab ;
end Tri_Bulles ;
```

Si vous faites quelques essais en affichant `Tab` à chaque itération, vous vous rendrez vite compte qu'avec cet algorithme le plus grand élément du tableau est immédiatement propagé à la fin du tableau dès le premier passage. Il serait donc plus

judicieuse de modifier les bornes de la deuxième boucle en écrivant :

Code : Ada

```
Tab'first .. Tab'last-1 - k
```

Où k correspondrait au nombre de fois que le tableau a été parcouru. Autre amélioration possible : puisque les déplacements des grands éléments vers la droite se font plus vite que celui des petits vers la gauche (déplacement maximal de 1 vers la gauche), pourquoi ne pas changer de sens une fois sur deux : parcourir le tableau de gauche à droite, puis de droite à gauche, puis de gauche à droite... c'est ce que l'on appelle le *Cocktail Shaker Sort*, un poil plus efficace.

Enfin, une dernière amélioration mérite d'être signalée (mais non traitée), c'est le *CombSort*. Il s'agit d'un tri à bulles nettement amélioré puisqu'il peut rivaliser avec les meilleurs algorithmes de tri. L'idée, plutôt que de comparer des éléments successifs, est de comparer des éléments plus éloignés afin d'éviter les *tortues*, c'est-à-dire ces petits éléments qui ne se déplacent que d'un seul cran vers la gauche avec le tri à bulles classique. Ce cours n'ayant pas vocation à rédiger tous les algorithmes de tri possibles et imaginables, consultez le lien ci-dessous ou [wikipedia](#) pour en savoir plus.

Algorithmes de tri plus rapides

Tri rapide (ou Quick Sort)

Principe

L'algorithme de tri rapide (ou Quick Sort) a un fonctionnement fort différent des algorithmes précédents et un peu plus compliqué (certains diront beaucoup plus). Le principe de base est simple mais bizarre a priori : on choisit une valeur dans le tableau appelée pivot (nous prendrons ici la première valeur du tableau) et on déplace avant elle toutes celles qui lui sont inférieures et après elle toutes celles qui lui sont supérieures. Puis, on réitère le procédé avec la tranche de tableau inférieure et la tranche de tableau supérieure à ce pivot.

Voici un exemple :

13	18	9	15	7
----	----	---	----	---

Nous prenons le premier nombre en pivot : 13 et nous plaçons les nombres 9 et 7 avant le 13, les nombres 15 et 18 après le 13.

9	7	13	15	18
---	---	----	----	----

Il ne reste plus ensuite qu'à réitérer l'algorithme sur les deux sous-tableaux.

9	7
15	18

Le premier aura comme pivot 9 et sera réorganisé, le second aura comme pivot 15 et ne nécessitera aucune modification. Il restera alors deux sous-sous-tableaux.

7
18

Comme ces sous-sous-tableaux se résument à une seule valeur, l'algorithme s'arrêtera. Cet algorithme est donc récursif et l'une des difficultés est de répartir les valeurs de part et d'autre du nombre pivot. Revenons, à notre tableau initial. Nous allons chercher le premier nombre plus petit que 13 de gauche à droite avec une variable i , et le premier nombre plus grand que 13 de droite à gauche avec une variable j , jusqu'à ce que i et j se rencontrent.

13	18	9	15	7
----	----	---	----	---

On trouve 18 et 7. On les inverse et on continue.

13	7	9	15	18
----	---	---	----	----

Le rangement est presque fini, il ne reste plus qu'à inverser le pivot avec le 9.

9	7	13	15	18
---	---	----	----	----

Pour aller plus loin ou pour davantage d'explication, vous pouvez consulter le [tutoriel de GUILoooo](#) à ce sujet (langages utilisés : C et OCaml).

Mise en œuvre

Cet algorithme va être plus compliqué à mettre en œuvre. Je vous conseille d'essayer par vous-même afin de faciliter la compréhension de la solution que je vous fournis :

Secret (cliquez pour afficher)

Code : Ada

```
function tri_rapide (T : vecteur) return vecteur is
  procedure tri_rapide(T : in out vecteur ; premier,dernier :
  integer) is
    index_pivot : integer := (premier+dernier)/2      ;
    j           : integer := premier + 1;
  begin
    if premier < dernier
      then echanger(T(premier),T(index_pivot)) ;
      index_pivot := premier ;
      for i in premier+1..dernier loop
        if T(i) < T(index_pivot)
          then echanger(T(i),T(j)) ;
          j := j + 1 ;
        end if ;
      end loop ;
      echanger(T(index_pivot),T(j-1)) ;
      index_pivot := j-1 ;
    tri_rapide(T, premier, Index_pivot-1) ;
    tri_rapide(T, Index_pivot+1, dernier) ;
  end if ;
  end tri_rapide ;
  Tab : Vecteur := T ;
begin
  tri_rapide(Tab,Tab'first,Tab'last) ;
  return Tab ;
end tri_rapide ;
```

Nous verrons un peu plus tard pourquoi j'ai utilisé une procédure plutôt qu'une fonction pour effectuer ce tri.

Tri fusion (ou Merge Sort)

Principe

Ah ! Mon préféré. Le principe est simple mais rudement efficace (un peu moins que le tri rapide toutefois). Prenons le tableau ci-dessous :

8	5	13	7	10
---	---	----	---	----

L'idée est de «scinder» ce tableau en cinq cases disjointes que l'on va «fusionner» ensuite, mais dans le bon ordre. Par exemple, les deux premières cases, une fois refusionnées donneront :

5	8
---	---

Les deux suivantes donneront :

7	13
---	----

Et la dernière va rester seule pour l'instant. Ensuite, on va «refusionner» nos deux minitableaux (la dernière case restant encore seule) :

5	7	8	10	13
---	---	---	----	----

Et enfin, on fusionne ce dernier tableau avec la dernière case restée seule depuis le début :

5	7	8	10	13
---	---	---	----	----

Vous faurez compris, le tri se fait au moment de la fusion des sous-tableaux. L'idée n'est pas si compliquée à comprendre, mais un peu plus ardue à mettre en œuvre. En fait, nous allons créer une fonction `tri_fusion` qui sera récursive et qui s'appliquera à des tranches de tableaux (les deux moitiés de tableaux, puis les quatre moitiés de moitiés de tableaux, puis ...) avant de les refusionner. Pour aller plus loin ou pour davantage d'explication, vous pouvez consulter le [tutoriel de renO](#) à ce sujet (langage utilisé : Python).

Mise en œuvre

Nous aurons besoin de deux fonctions pour effectuer le tri fusion : une fonction `fusion` qui fusionnera deux tableaux triés en un seul tableau trié ; et une fonction `tri_fusion` qui se chargera de scinder le tableau initial en sous-tableaux triés avant de les refusionner à l'aide de la précédente fonction.

Secret (cliquez pour afficher)

```
Code : Ada
```

```
function fusion(T1,T2 : Vecteur) return Vecteur is
  T : Vecteur(1..T1'length + T2'length);
  i : integer := T1'first;
  j : integer := T2'first;
begin
  if T1'length = 0 then return T2;
  elsif T2'length = 0 then return T1;
  end if;
  for k in T'range loop
    if i > T1'last and j > T2'last then exit;
    elsif i = T1'last then T(k..T1'last) := T2(j..T2'last); exit;
    elsif j > T2'last then T(k..T1'last) := T1(i..T1'last); exit;
    end if;
    if T1(i) <= T2(j) then
      T(k) := T1(i);
      i := i + 1;
    else T(k) := T2(j);
      j := j + 1;
    end if;
  end loop;
  return T;
end fusion;

FUNCTION Tri_fusion(T: Vecteur) RETURN Vecteur IS
  lg : constant Integer := T'length;
BEGIN
  if lg <= 1 then return T;
  else declare
    T1 : Vecteur(T'first..T'first+lg/2);
    T2 : Vecteur(T'first+lg/2..T'last);
    begin
      T1 := tri_fusion(T(T'first..T'first+lg/2));
      T2 := tri_fusion(T(T'first+lg/2..T'last));
      return fusion(T1,T2);
    end;
  end if;
END Tri_Fusion;
```

Tri par tas

Noms alternatifs

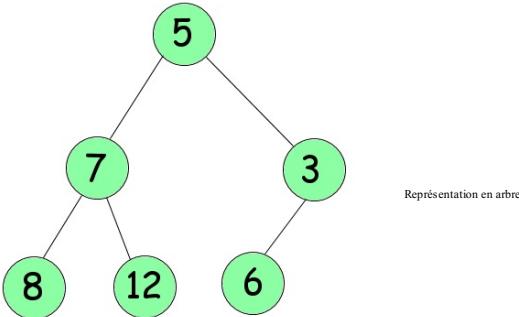
Le tri par tas porte également les noms suivants :

- Tri arbre
- Tri Maximier
- Heapsort
- Tri de Williams

Principe

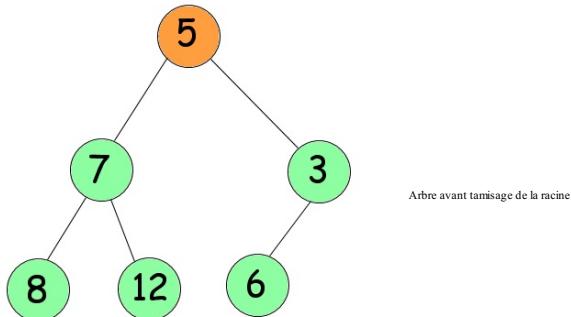
5	7	3	8	12	6
---	---	---	---	----	---

L'idée du tri par tas est de concevoir votre tableau tel un **arbre**. Par exemple, le tableau ci-dessus devra être vu ainsi :

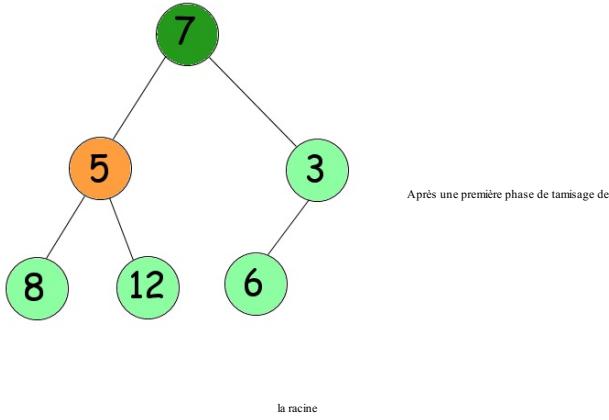


Chaque «case» de l'arbre est appelée **noeud**, le premier noeud porte le nom de **racine**, les derniers le nom de **feuille**. Si l'on est rendu au noeud numéro n (indice dans le tableau initial), ses deux feuilles porteront les numéros $2n$ et $2n + 1$.

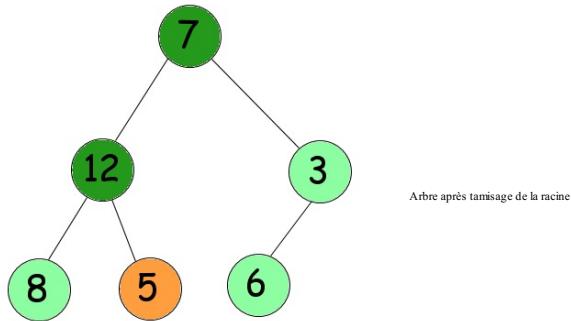
L'arbre n'est pas trié pour autant. Pour cela nous devrons le **tamiser**, c'est-à-dire prendre un nombre et lui faire descendre l'arbre jusqu'à ce que les nombres en dessous lui soient inférieurs. Pour l'exemple, nous allons tamiser le premier noeud (le 5).



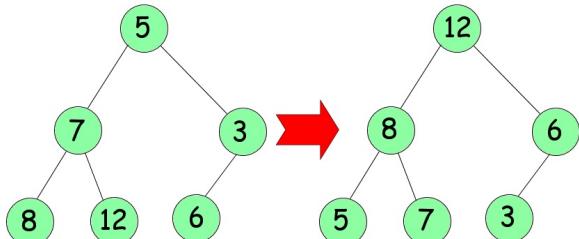
En dessous de lui se trouvent deux noeuds : 7 et 3. On choisit le plus grand des deux : 7. Comme 7 est plus grand que 5, on les inverse.



On recommence : les deux noeuds situés en dessous sont 8 et 12. Le plus grand est 12. Comme 12 est plus grand que 5, on les inverse.

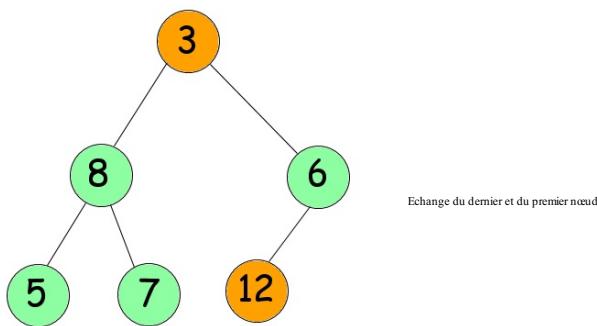


Vous avez compris le principe du tamisage ? Très bien. Pour trier notre arbre par tas, nous allons le tamiser en commençant, non pas par la racine de l'arbre, mais par les derniers noeuds (en commençant par la fin en quelque sorte). Bien sûr, il est inutile de tamiser les feuilles de l'arbre : elles ne descendentront pas plus bas ! Pour limiter les tests inutiles, nous commencerois par tamiser le 3, puis le 7 et enfin le 5. Du coup, nous ne tamiserons que la moitié des noeuds de l'arbre. Faites l'essai, cela devrait vous donner ceci :

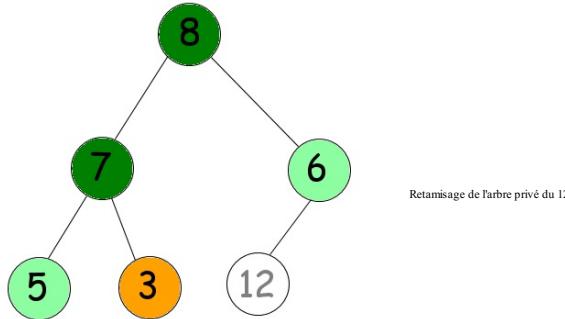


Hein ??? Plus on descend dans l'arbre plus les nombres sont petits, mais c'est pas trié pour autant !???

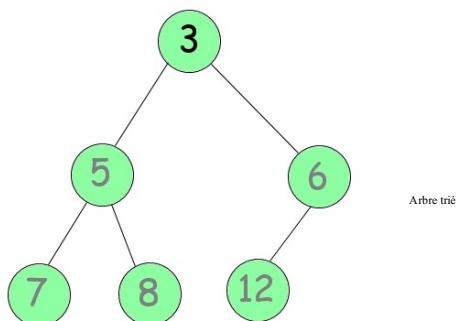
En effet. Mais rassurez-vous, il reste encore une étape, un peu plus complexe. Nous allons parcourir notre arbre en sens inverse encore une fois. Nous allons échanger le dernier noeud de notre arbre tamisé (le 3) avec le premier (le 12) qui est également le plus grand nombre de l'arbre.



Une fois cet échange effectué, nous allons tamiser notre nouvelle racine, en considérant que la dernière feuille ne fait plus partie de l'arbre (elle est correctement placée désormais). Ce qui devrait nous donner le résultat suivant (voir la figure suivante).



Cette manœuvre a ainsi pour but de placer sur la dernière feuille, le nombre le plus grand, tout en gardant un sous-arbre qui soit tamisé. Il faut ensuite répéter cette manœuvre avec l'avant-dernière feuille, puis l'avant-avant-dernière feuille... Jusqu'à arriver à la deuxième (et oui, inutile d'échanger la racine avec elle-même, soyons logique). Nous devrions ainsi obtenir le résultat suivant (voir la figure suivante).



Cette fois, l'arbre est trié dans le bon ordre, l'algorithme est terminé. Il ne vous reste plus qu'à l'implémenter.

Mise en œuvre

Vous faurez deviné je l'espère, il nous faut une procédure tamiser, en plus de notre fonction Tri_tas. Commencez par la créer et l'essayer sur le premier élément comme nous l'avons fait dans cette section avant de vous lancer dans votre fonction de tri. Pensez qu'il doit être possible de limiter la portion de tableau à tamiser et que pour passer d'un nœud n à celui du dessous, il faut multiplier n par 2 (pour accéder au nœud en dessous à gauche) et éventuellement lui ajouter 1 (pour accéder à celui en dessous à droite).

Secret (cliquez pour afficher)

Code : Ada

```

procedure tamiser(T : in out vecteur ;
                   noeud : integer ;
                   max : integer) is
    racine : integer := noeud ;
    feuille : integer := 2 * noeud ;
begin
    while feuille <= max loop
        if feuille+1 <= max and then T(feuille)<T(feuille+1)
        then feuille := feuille + 1;
        end if;
        if T(racine)<T(feuille)
        then echanger(T(racine), T(feuille));
        end if;
        racine := feuille;
        feuille := 2 * racine;
    end loop;
end tamiser;

function tri_tas(T : vecteur) return vecteur is
    Arbre : vecteur(1..T'length) := T ;
begin
    for i in reverse 1..Arbre'length/2 loop
        tamiser(Arbre,i,Arbre'length);
    end loop;
    for i in reverse 2..Arbre'last loop
        echanger(Arbre(i),Arbre(1));
    end loop;
end tri_tas;

```

```

        tamiser(Arbre,1,i-1) ;
    end loop ;
return Arbre ;
end tri_tas ;

```

Théorie : complexité d'un algorithme

Complexité

Si vous avez dès lors déjà testé les programmes ci-dessus sur de grands tableaux, vous avez du vu vous rendre compte que certains étaient plus rapides que d'autres, plus efficaces. C'est ce que l'on appelle **l'efficacité** d'un algorithme. Pour la mesurer, la quantifier, on s'intéresse davantage à son contrepartie, la **complexité**. En effet, pour mesurer la complexité d'un algorithme il « suffit » de mesurer la quantité de ressources qu'il exige (mémoire ou temps-processeur).

En effet, le simple fait de déclarer un tableau exige de réquisitionner des emplacements mémoires qui seront rendus inutilisables par d'autres programmes comme votre système d'exploitation. Tester l'égalité entre deux variables va nécessiter de réquisitionner temporairement le processeur, le rendant très brièvement inutilisable pour toute autre chose. Vous comprenez alors que parcourir un tableau de 10 000 cases en testant chacune des cases exige donc de réquisitionner au moins 10 000 emplacements mémoires et d'interrompre 10 000 fois le processeur pour faire nos tests. Tout cela ralentit les performances de l'ordinateur et peut même, pour des algorithmes mal ficelés, saturer la mémoire entraînant l'erreur **STORAGE_ERROR** : **EXCEPTION_STACK_OVERFLOW**.

D'où l'intérêt de pouvoir comparer la complexité de différents algorithmes pour ne conserver que les plus efficaces, voire même de prédir cette complexité. Prenons par exemple l'algorithme dont nous parlions précédemment (qui lit un tableau et teste chacune de ses cases) : si le tableau a 100 cases, l'algorithme effectuera 100 tests ; si le tableau à 5000 cases, l'algorithme effectuera 5000 tests ; si le tableau a n cases, l'algorithme effectuera n tests ! On dit que sa complexité est en $O(n)$.

L'écriture O



Ça veut dire quoi ce zéro ?

Ce n'est pas un zéro mais la lettre O ! Et $O(n)$ se lit « grand O de n ». Cette notation mathématique signifie que la complexité de l'algorithme est proportionnelle à n (le nombre d'éléments contenus dans le tableau). Autrement dit, la complexité vaut « grosso-modo » n .



C'est pas « grosso modo égal à n » ! C'est exactement égal à n ! ☺

Prenons un autre exemple : un algorithme qui parcourt lui aussi un tableau à n éléments. Pour chaque case, il effectue deux tests : le nombre est-il positif ? Le nombre est-il pair ? Combien va-t-il effectuer de tests ? La réponse est simple : deux fois plus que le premier algorithme, c'est à dire $2 \times n$. Autrement dit, il est deux fois plus lent. Mais pour le mathématicien ou l'informaticien, ce facteur 2 n'a pas une importance aussi grande que cela, l'algorithme a toujours une complexité proportionnelle au premier, et donc en $O(n)$



Mais alors tous les algorithmes ont une complexité en $O(n)$ si c'est comme ça ?!

Non, bien sûr. Un facteur 2, 3, 20... peut être considéré comme négligeable. En fait, tout facteur constant peut être considéré comme négligeable. Reprenons encore notre algorithme : il parcourt toujours un tableau à n éléments, mais à chaque case du tableau, il repartout tout le tableau depuis le début pour savoir s'il n'y aurait pas une autre case ayant la même valeur. Combien va-t-il faire de tests ? Pour chaque case, il doit faire n tests et comme il y a n cases, il devra faire en tout $n \times n = n^2$ tests. Le facteur n'est cette fois pas constant ! On dira que cet algorithme a une complexité en $O(n^2)$ et, vous vous en doutez, ce n'est pas terrible du tout. Et pourtant, c'est la complexité de certains de nos algorithmes de tri ! ☺

Donc, vous devez commencer à comprendre qu'il est préférable d'avoir une complexité en $O(n)$ qu'en $O(n^2)$. Malheureusement pour nous, en règle générale, il est impossible d'atteindre une complexité en $O(n)$ pour un algorithme de tri (sauf cas particulier comme avec un tableau déjà trié, et encore ça dépend des algorithmes ☺). Un algorithme de tri aura une complexité optimale quand celle-ci sera en $O(n \times \ln(n))$

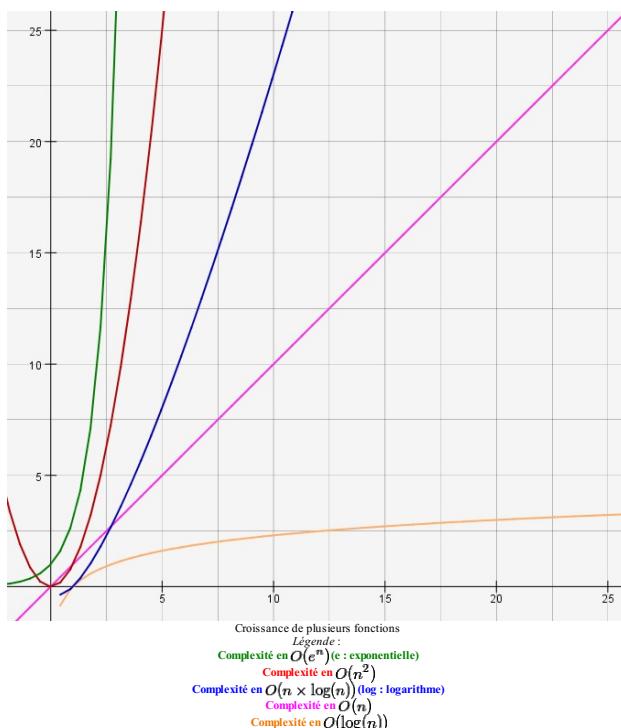
Quelques fonctions mathématiques



Qu'est-ce que c'est encore que cette notation $\ln(n)$?

C'est une fonction mathématique que l'on appelle le **logarithme népérien**. Les **logarithmes** sont les fonctions mathématiques qui ont la croissance la plus faible (en fait elles croissent très vite au début puis ralentissent par la suite). Inversement, les fonctions **exponentielles** sont celles qui ont la croissance la plus rapide (très très lentes au début puis de plus en plus rapide).

Mon but ici n'est pas de faire de vous des mathématiciens, encore moins d'expliquer en détail les notions de fonction ou de « grand O », mais pour que vous ayez une idée de ces différentes fonctions, de leur croissance et surtout des complexités associées, je vous en propose quelques-unes sur le graphique ci-dessous :



Plus l'on va vers la droite, plus le nombre d'éléments à trier, tester, modifier... augmente. Plus l'on va vers le haut, plus le nombre d'opérations effectuées augmente et donc plus la quantité de mémoire ou le temps processeur nécessaire augmente. On voit ainsi qu'une complexité en $O(\log(n))$ ou $O(\ln(n))$ c'est peu ou prou pareil, serait parfaite (on parle de complexité logarithmique) : elle augmente beaucoup moins vite que le nombre d'éléments à trier ! Le rêve. Sauf que ça n'existe pas pour nos algorithmes de tri, donc oublez-la.

Une complexité en $O(n)$ serait donc idéale (on parle de complexité linéaire). C'est possible dans des cas très particuliers : par exemple l'algorithme de tri par insertion peut atteindre une complexité en $O(n)$ lorsque le tableau est déjà trié ou « presque trié », alors que les algorithmes de tri rapide ou fusion n'en sont pas capables. Mais dans la vie courante, il est rare que l'on demande de trier des tableaux déjà triés.

On comprend donc qu'atteindre une complexité en $O(n \times \ln(n))$ est déjà satisfaisant (on parle de complexité linéarithmique), on dit alors que la **complexité est asymptotiquement optimale**, bref, on ne peut pas vraiment faire mieux dans la majorité des cas.

Une complexité en $O(n^2)$ dite complexité quadratique, est courante pour des algorithmes de tri simples (ou simplistes), mais très mauvaise. Si au départ il ne diffère pas beaucoup d'une complexité linéaire (en $O(n)$), elle finit par augmenter très vite à mesure que le nombre d'éléments à trier augmente. Donc les algorithmes de tri de complexité quadratique seront réservés pour de petits tableaux.

Enfin, la complexité exponentielle (en $O(e^n)$) est la pire chose qui puisse vous arriver. La complexité augmente à vite grand V, de façon exponentielle (logique me direz-vous). N'oubliez qu'un seul élément et vous aurez beaucoup plus d'opérations à effectuer. Pour un algorithme de tri, c'est le signe que vous coderez comme un cochon. Attention, les jugements émis ici ne concernent que les algorithmes de tri. Pour certains problèmes, vous serez heureux d'avoir une complexité en seulement $O(n^2)$.

Mesures de complexité des algorithmes

Un algorithme pour mesurer la complexité... des algorithmes

Après cette section très théorique, retournons à la pratique. Nous allons rédiger un programme qui va mesurer le temps d'exécution des différents algorithmes avec des tableaux de plus en plus grands. Ces temps seront enregistrés dans un fichier texte de manière à pouvoir ensuite transférer les résultats dans un tableau (Calc ou Excel par exemple) et dresser un graphique. Nous en déduirons ainsi, de manière empirique, la complexité des différents algorithmes. Pour ce faire, nous allons de nouveau faire appel au package **Ada.Calendar** et effectuer toute une batterie de tests à répétition avec nos divers algorithmes. Le principe du programme sera le suivant :

Code : Français

```
POUR i ALLANT DE 1 A 100 REPETER
|   Créer Tableau Aléatoire de taille i*100
|   Mesurer Tri_Selection(Tableau)
|   Mesurer Tri_Insertion(Tableau)
|   Mesurer Tri_Bulles(Tableau)
|   Mesurer Tri_Rapide(Tableau)
|   Mesurer Tri_Fusion(Tableau)
|   Mesurer Tri_Tas(Tableau)
|   Afficher "Tests effectués pour taille" i*100
|   incrémenter i
FIN BOUCLE
```

La boucle devrait effectuer 100 tests, ce qui sera largement suffisant pour nos graphiques. Nous ne dépasserons pas des tableaux de taille 10 000 (les algorithmes lents devraient déjà mettre de 1 à 4 secondes pour effectuer leur tri), car au-delà, vous pourriez vous lasser. Par exemple, il m'a fallu 200s pour trier un tableau de taille 100 000 avec le tri par insertion et 743s avec le tri à bulles alors qu'il ne faut même pas une demi-seconde aux algorithmes rapides pour faire ce travail. Si vous le souhaitez, vous pourrez reprendre ce principe avec des tableaux plus importants pour les algorithmes plus rapides (vous pouvez facilement aller jusqu'à des tableaux de taille 1 million).



Mais que vont faire ces procédures ?

C'est simple, elles doivent :

1. ouvrir un fichier texte ;
2. lire "lheure" ;
3. exécuter l'algorithme de tri correspondant ;
4. lire à nouveau "lheure" pour en déduire la durée d'exécution ;
5. enregistrer la durée obtenue dans le fichier ;
6. fermer le fichier.

Cela impliquera de créer un type de pointeur sur les fonctions de tri. Voici le code source du programme :

Secret (cliquez pour afficher)

Code : Ada

```
procedure sort is
type T_Ptr_Fonction is access function(T : Vecteur) return Vecteur;
type T_Tri is (SELECTION, INSERTION, BULLES, FUSION, RAPIDE, TAS);
procedure mesurer(tri : T_Ptr_Fonction; T : Vecteur; choix : T_Tri) is
    temps : time;
    duree : duration;
    U :Vecteur(T'range);
    F : File_Type;
begin
    case choix is
        when SELECTION =>
            open(F,Append_file,"./mesures/selection.txt");
        when INSERTION =>
            open(F,Append_file,"./mesures/insertion.txt");
        when BULLES => open(F,Append_file,"./mesures/bulles.txt");
        when FUSION => open(F,Append_file,"./mesures/fusion.txt");
        when RAPIDE => open(F,Append_file,"./mesures/rapide.txt");
        when TAS => open(F,Append_file,"./mesures/tas.txt");
    end case;
    temps := clock;
    U := Tri(T);
    duree := clock - temps;
    put(F,float(duree),Exp=>0);
    new_line(F);
    close(F);
end mesurer;
tri : T_Ptr_Fonction;
F : File_Type;
begin
create(F,Append_file,"./mesures/selection.txt"); close(F);
create(F,Append_file,"./mesures/insertion.txt"); close(F);
create(F,Append_file,"./mesures/bulles.txt"); close(F);
create(F,Append_file,"./mesures/fusion.txt"); close(F);
create(F,Append_file,"./mesures/rapide.txt"); close(F);
create(F,Append_file,"./mesures/tas.txt"); close(F);

for i in 1..100 loop
    declare
        T : Vecteur(1..i*100);
    begin
        generate(T,0,i*50);
        tri := Tri select'access;
        mesurer(tri,T,SELECTION);
        tri := Tri insertion'access;
        mesurer(tri,T,INSERTION);
        tri := Tri bulles'access;
        mesurer(tri,T,BULLES);
        tri := Tri fusion'access;
        mesurer(tri,T,FUSION);
        tri := Tri rapide'access;
        mesurer(tri,T,RAPIDE);
        tri := Tri_tas'access;
        mesurer(tri,T,TAS);
    end;
    put("Tests effectués pour la taille"); put(i*100);
    new_line;
end loop;
end sort;
```



L'exécution de ce programme nécessitera quelques minutes. Ne vous inquiétez pas si sa vitesse d'exécution ralentit au fur et à mesure : c'est normal puisque les tableaux sont de plus en plus lourds.

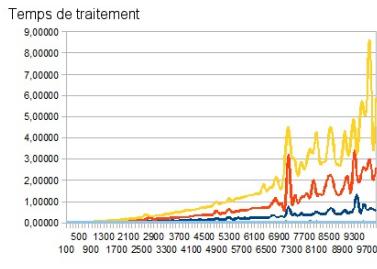
Traiter nos résultats

À l'aide du programme précédent, vous devez avoir obtenu six fichiers textes contenant chacun 100 valeurs décimales.



Et j'en fais quoi maintenant ? Je vais tout comparer à la main !

Bien sûr que non. Ouvrez donc un tableau (tel Excel ou Calc d'OpenOffice) et copiez-y les résultats obtenus. Sélectionnez les valeurs obtenues puis cliquez sur Insertion->Diagramme afin d'obtenir une représentation graphique des résultats obtenus. Je vous présente sur la figure suivante les résultats obtenus avec mon vieux PC.



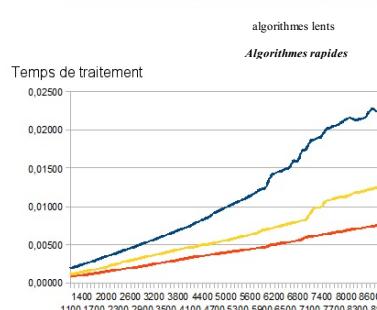
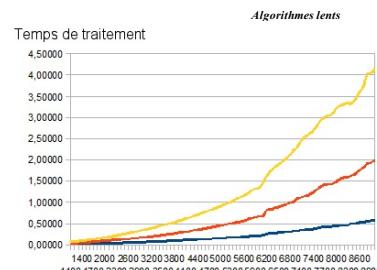
Avant même de comparer les performances de nos algorithmes, on peut remarquer une certaine irrégularité : des pics et des creux apparaissent et semblent (en général) avoir lieu pour tous les algorithmes. Ces pics et ces creux correspondent aux pires et aux meilleurs cas possibles. En effet, si notre programme de génération aléatoire de tableaux vient à créer un tableau quasi-trié, alors le temps mis par certains algorithmes chuterait drastiquement. Inversement, certaines configurations peuvent être réellement problématiques à certains algorithmes : prenez un tableau quasi-trié mais dans le mauvais ordre ! Ce genre de tableau est catastrophique pour un algorithme de tri à bulles qui devra effectuer un nombre de passages excessif : il suffit, pour s'en convaincre, de regarder le dernier pic de la courbe jaune ci-dessus (tri à bulles) ; d'ailleurs ce pic est nettement atténué sur la courbe orange (tri par insertion) et quasi inexistant sur la courbe bleu (tri par sélection).

Qui plus est, il apparaît assez nettement que le temps de traitement des algorithmes rapides (tri rapide, tri fusion et tri par tas) est quasiment négligeable par rapport à celui des algorithmes lents (tri par insertion, tri par sélection et tri à bulles), au point que ceux-ci sont presque invisibles sur le graphique.

Pour y voir plus clair, je vous propose de traiter nos chiffres avec un peu plus de finesse : nous allons distinguer les algorithmes lents des algorithmes rapides et nous allons lisser nos résultats. Par liser, j'entends modifier nos résultats pour ne plus être gênés par ces pics et ces creux et voir apparaître une courbe plus agréable représentant la tendance de fond.

Secret ([cliquez pour afficher](#))

Lissage des résultats par moyenne mobile : Pour ceux qui se demanderaient ce qu'est ce lissage par moyenne mobile, voici une explication succincte (ce n'est pas un cours de statistiques non plus). Au lieu de traiter nos 100 nombres, nous allons créer une nouvelle liste de valeurs. La première valeur est obtenue en calculant la moyenne des 20 premiers nombres (du 1er au 20ème), la seconde est obtenue en calculant la moyenne des nombres entre le 2nd et le 21ème, la troisième valeur est obtenue en calculant la moyenne des nombres entre le 3ème et le 22ème... La nouvelle liste de valeurs est ainsi plus homogène mais aussi plus courte (il manquera 20 valeurs).



Ainsi retravaillés, nos résultats sont plus lisibles et on voit apparaître qu'en moyenne, l'algorithme le plus efficace est le tri rapide, puis viennent le tri par tas et le tri fusion. Mais même le tri fusion reste, en moyenne, en dessous de 0,025 seconde pour un tableau à 10 000 éléments sur mon vieil ordinateur (son score est sûrement encore meilleur chez vous), ce qui est très loin des performances des trois autres algorithmes. Pour les trois plus lents, remarquez que le « pire » d'entre eux est, en moyenne, le tri à bulles, puis viennent le tri par insertion et le tri par sélection (avec des écarts de performance toutefois très nets entre eux trois).

Le lissage par moyenne mobile nous permet également de voir se dégager une tendance, une courbe théorique débarrassée des impuretés (pire cas et meilleur cas). Si l'on considère les algorithmes lents, on distingue un début de courbe en U, même si le U semble très écrasé par l'algorithme de tri par sélection. Cela nous fait donc penser à une complexité en $O(n^2)$ où n est le nombre d'éléments du tableau.



Et pourquoi pas en $O(e^n)$?

Prenez un tableau à n éléments que l'on voudrait trier avec le pire algorithme : le tri à bulles. Dans le pire des cas, le tableau est trié en sens inverse et le plus petit élément est donc situé à la fin. Et comme le tri à bulle ne déplace le plus petit élément que d'un seul cran à chaque passage, il faudra donc effectuer $n - 1$ passages pour que ce minimum soit à sa place et le tableau trié. De plus, à chaque itération, l'algorithme doit effectuer $n - 1$ comparaison (et au plus $n - 1$ échanges). D'où un total de $(n - 1) \times (n - 1)$ comparaisons. Or $(n - 1) \times (n - 1) = n^2 - 2 \times n + 1$, donc la complexité est « de l'ordre de » n^2 . Dans le pire cas, l'algorithme tri à bulles a une complexité en $O(n^2)$. Cette famille d'algorithme a donc une complexité quadratique et non exponentielle.

Considérons maintenant nos trois algorithmes rapides, leur temps d'exécution est parfaitement négligeable à côté des trois précédents, leurs courbes croissent peu : certaines semblent presque être des droites. Mais ne vous y trompez pas, leur complexité n'est sûrement pas linéaire, ce serait trop beau. Elle est toutefois linéarithmique c'est-à-dire en $O(n \times \ln(n))$. Ces trois algorithmes sont donc asymptotiquement optimaux.



Différentes variantes de ces algorithmes existent. Les qualificatifs de «lent», «rapide», «pire» ou «meilleur» sont donc à prendre avec des pincettes, car certains peuvent être très nettement améliorés ou sont très utiles dans des cas particuliers.

Par exemple, notre algorithme de tri fusion est un tri en place : nous n'avons pas créé de second tableau pour effectuer notre tri afin d'économiser la mémoire, denrée rare si l'on trie de très grands tableaux (plusieurs centaines de millions de cases). Une variante consiste à ne pas l'effectuer en place : on crée alors un second tableau et l'opération de fusion gagne en rapidité. Dans ce cas, l'algorithme de tri fusion acquiert une complexité comparable à celle du tri rapide en réalisant moins de comparaisons, mais devient gourmand en mémoire.

De la même manière, nous avons évoqué le [CombSort](#) qui est une amélioration du tri à bulles et qui permet à notre «pire» algorithme de cotoyer les performances des «meilleurs». Encore un dernier exemple, le «lent» algorithme de tri par insertion atteint une complexité linéaire en $O(n)$ sur des tableaux quasi-très alors que l'algorithme de tri rapide garde sa complexité linéaire en $O(n \times \ln(n))$.

En résumé :

- De nombreux domaines ont besoin d'algorithmes efficaces : traitement du génome humain, météorologie, positionnement par satellite, opérations financières, calcul intensif... Vous devez donc vous interroger sur l'efficacité des algorithmes que vous rédigez.
- La notion d'efficacité n'est pas toujours évidente : un algorithme simple peut se révéler très gourmand, un « mauvais » algorithme peut décrocher une médaille d'or dans des conditions particulières ou moyennant quelques innovations.
- N'hésitez pas à effectuer des essais grandeur nature avec de grandes quantités de données ou à compter le nombre d'opérations et de tests nécessaires pour juger de la pertinence de votre méthode.
- J'espère que ce chapitre vous aura fait prendre conscience de l'importance des Mathématiques en Informatique et notamment que l'Algorithmique en constitue une branche à part entière, peut-être la plus complexe.

Variables III : Gestion bas niveau des données

Le chapitre que nous allons aborder est très théorique et traite de diverses notions ayant trait à la mémoire de l'ordinateur : comment est-elle gérée ? Comment les informations y sont-elles représentées ? Etc. Nous allons passer le plus clair du chapitre à ne pas parler de programmation (étrange me direz-vous pour un cours de programmation !) mais ce n'est que pour mieux revenir sur les types Integer, Character et Float, sur de nouveaux types (Long_Long_Float, Long_Long_Integer...) ou sur certaines erreurs vues au fil du cours.

Chaque partie de ce chapitre abordera la manière dont l'ordinateur représente les nombres entiers, décimaux ou les caractères en mémoire. Ce sera l'occasion de parler binaire, hexadécimal, virgule flottante... avant de soulever certains problèmes liés à ces représentations et d'en déduire quelques bonnes pratiques.

Représentation des nombres entiers

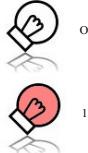
Le code binaire

Nous allons essayer, au cours des sections suivantes, de comprendre comment l'ordinateur s'y prend pour représenter les nombres entiers, les caractères ou les nombres décimaux, car cela a des implications sur la façon de programmer. En effet, depuis le début, nous faisons comme s'il était normal que l'ordinateur sache compter de -2 147 483 648 à 2 147 483 647, or je vous rappelle qu'un ordinateur n'est jamais qu'un amas de métal et de silicium. Alors comment peut-il compter ? Et pourquoi ne peut-il pas compter au-delà de 2 147 483 647 ? Que se passe-t-il si l'on va au-delà ? Comment aller au-delà ? Reprenons tout à la base.

Tout d'abord, un ordinateur fonctionne à l'électricité (jusque là rien de révolutionnaire) et qu'avec cette électricité nous avons deux états possibles : éteint ou allumé.

 Eh ! Tu me prends pour un idiot ou quoi ? 😂

Bien sûr que non, mais c'est grâce à ces deux états de l'électricité que les Hommes ont pu créer des machines aussi perfectionnées capables d'envoyer des vidéos à l'autre bout du monde. En effet, à chaque état peut-être associé un nombre :



Lorsque le courant passe, cela équivaut au chiffre 1, lorsqu'il ne passe pas, ce chiffre est 0. De même lorsqu'un accumulateur est chargé, cela signifie qu'un 1 est en mémoire, s'il est déchargé, le chiffre 0 est enregistré en mémoire. Nos ordinateurs ne disposent donc que de deux chiffres 0 et 1. Nous comptons quant à nous avec 10 chiffres (0, 1, 2, ..., 8, 9) : on dit que nous comptons en **base 10** ou **décimale**. Les ordinateurs comptent donc en **base 2**, aussi appelé **binaire**. Les chiffres 1 et 0 sont appelés **bits**, pour binary digits c'est à dire chiffre binaire.

 Mais, comment faire un 2 ou un 3 ?

Il faut procéder en binaire comme nous le faisons en base 10 : lorsque l'on compte jusqu'à 9 et que l'on a épousé tous nos chiffres, que fait-on ? On remet le chiffre des unités à 0 et on augmente la dizaine. Eh bien c'est pareil pour le binaire : après 0 et 1, nous aurons donc 10 puis 11. On peut ensuite ajouter une «centaine» et recommencer. Voici une liste des premiers nombres en binaire (je vous invite à essayer de la compléter par vous-même, ce n'est pas compliqué) :

Décimal	Binaire
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010

Conversions entre décimal et binaire

Vous aurez remarqué les répétitions : pour les bits des «unités», on écrit un 0 puis un 1, puis un 0... pour les bits des «dizaines», on écrit deux 0 puis deux 1, puis deux 0... pour les bits des «centaines», on écrit quatre 0, puis quatre 1, puis quatre 0... et ainsi de suite en doublant à chaque fois le nombre de 0 ou de 1. Mais il y a une ombre au tableau. Nous n'allons pas faire un tableau allant jusqu'à 2 147 483 647 pour pouvoir le convertir en binaire ! Nous devons trouver un moyen de convertir rapidement un nombre binaire en décimal et inversement.

Conversion binaire vers décimal

Imaginons que nous recevions le signal suivant :



Représentation d'un message binaire

Ce message correspond au nombre binaire 10110 qui se décompose en 0 «unités», 1 «dizaine», 1 «centaine», 0 «millier» et 1 «dizaine de millier». Ce qui s'écrirait normalement :



$$\begin{aligned} 10110 &= (1 \times 10000) + (0 \times 1000) + (1 \times 100) + (1 \times 10) + (0 \times 1) \\ 10110 &= (1 \times 10^4) + (0 \times 10^3) + (1 \times 10^2) + (1 \times 10^1) + (0 \times 10^0) \end{aligned}$$

Sauf qu'en binaire, on ne compte pas avec dix chiffres mais avec seulement deux ! Il n'y a pas de «dizaines» ou de «centaines» mais plutôt des «deuxaines», «quatraines», «huitaines»... Donc en binaire notre nombre correspond en fait à :

$$10110 : (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$$

Ainsi, il suffit donc d'effectuer ce calcul pour trouver la valeur en base 10 de 10110 et de connaître les puissances de deux :

$$10110 : (1 \times 16) + (0 \times 8) + (1 \times 4) + (1 \times 2) + (0 \times 1) = 16 + 4 + 2 = 22$$

Autre méthode pour effectuer la conversion, il suffit de dresser le tableau ci-dessous. Dans la première ligne, on écrit les puissances de 2 : 1, 2, 4, 8, 16... Dans la seconde ligne, on écrit les bits :

16	8	4	2	1
1	0	1	1	0

Il y va ainsi une fois 2, plus une fois 4 plus une fois 16 (et 0 fois pour le reste), d'où 22 !

Conversion décimal vers binaire

Imaginez, cette fois que nous souhaitons envoyer en message le nombre 13. Pour cela, nous allons dresser le même tableau que précédemment :

8	4	2	1
1	0	1	0



Et posons-nous la question : dans 13 combien de fois 8 ? Une fois bien-sûr ! Et il reste 5. D'où le tableau suivant :

8	4	2	1
1	-	-	-

Maintenant, recommençons ! Dans 5 combien de fois 4 ? Une fois et il reste 1 !

8	4	2	1
1	1	-	-

Dans 1 combien de fois 2. Zéro fois, logique. Et dans 1 combien de fois 1 : 1 fois.

8	4	2	1
1	1	0	1

Donc le nombre 13 s'écrit 1101 en binaire. Le tour est joué.

Retour sur le langage Ada

Représentation des Integer en Ada

Vous avez compris le principe ? Si la réponse est encore non, je vous conseille de vous entraîner à faire diverses conversions avant d'attaquer la suite, car nous n'en sommes qu'au B-A-BA. En effet, les ordinateurs n'enregistrent pas les informations bits par bits, mais plutôt par paquets. Ces paquets sont appelés **octets**, c'est-à-dire des paquets de 8 bits (même s'il est possible de faire des paquets de 2, 4 ou 9 bits).

Pour enregistrer un Integer, le langage Ada utilise non pas un seul mais quatre octets, c'est-à-dire $4 \times 8 = 32$ bits ! Avec un seul bit, on peut coder deux nombres (0 et 1), avec deux bits on peut coder quatre nombres (0, 1, 2, 3), avec trois bits on peut coder huit nombres... avec 32 bits, on peut ainsi coder 2^{32} nombres, soit 4 294 967 296 nombres.

Mais la difficulté ne s'arrête pas là. En effet, le plus grand Integer n'est pas 4 294 967 296, mais 2 147 483 647 (c'est-à-dire presque la moitié). Pourquoi ? Tout simplement parce qu'il faut partager ces 4 294 967 296 nombres en deux : les positifs et les négatifs. Comment les distinguer ?

La réponse est simple : tous les Integer commençant par un 0 seront positifs (**exemple : 00101001 10011101 10100011 11111001**) ; tous les integer commençant par un 1 seront négatifs (**exemple : 10101001 10011101 10100011 11111001**). Ce premier bit est appelé **bit de poids fort**. Facile hein ? Eh bien non !

Car il y a encore un problème : on se retrouve avec deux zéros !

C'est dommage et cela complique les opérations d'un point de vue électronique. Du coup, il faut décaler les nombres négatifs d'un cran : le nombre **10000000 00000000 00000000 00000000** correspondra non pas à **-0** mais à **-1**. Le premier 1 signifie que le nombre est négatif, les 31 zéros, si on les convertit en base 10, devraient donner 0, mais avec ce décalage, on doit ajouter 1, ce qui nous donne -1 !

Conséquence immédiate : l'Overflow et les types long

OUF ! C'est un peu compliqué tout ça... Et ça sert à quoi ?

Patience, je vous entraîne peu à peu dans les méandres de l'informatique afin de mieux revenir sur notre sujet : le langage Ada et la programmation. Cette façon de coder les nombres entiers, qui n'est pas spécifique au seul langage Ada, a deux conséquences. Tout d'abord le plus grand Integer positif est $2^{31} - 1$ quand le plus petit Integer négatif est -2^{31} : il y a un décalage de 1 puisque 0 est codé comme un positif et pas comme un négatif par la machine.

Deuxième conséquence, nous n'avons que 2^{32} nombres à notre disposition, dont le plus grand est seulement $2^{31} - 1$. C'est très largement suffisant en temps normal, mais pour des calculs plus importants, cela peut s'avérer un peu juste (souvenez-vous seulement des **calculs de factorielles**). En effet, si vous dépassez ce maximum, que se passera-t-il ? Eh bien plusieurs possibilités. Soient vous y allez façon «déménageur» en rédigeant un code faisant clairement à des nombres trop importants :

Code : Ada

```
n := 2**525;
p := Integer'last + 1;
```

Et alors, le compilateur vous arrêtera tout de suite par un doux message : **value not in range of type "Standard.Integer"**. Ou bien vous y allez de façon «masquée» (), le compilateur ne détectant rien (une boucle itérative ou récursive, un affichage du type Put(**Integer'last+1**)). Vos nombres devenant trop grands, ils ne peuvent plus être codés sur seulement quatre octets, il y a ce que l'on appelle un **dépassement de capacité** (aussi appelé **overflow** [j'adore ce mot !]). Vous vous exposez alors à deux types d'erreurs : soit votre programme plante en affichant une jolie erreur du type suivant.

Code : Console

```
raised CONSTRAINT_ERROR : ###.adb:# overflow check failed
```

Soit il ne détecte pas l'erreur (c'est ce qui est arrivé avec notre programme de **calcul de factorielles**) : le nombre positif devient trop grand, il «déborde» sur le bit de poids fort normalement réservé au signe, ce qui explique que nos factorielles devaient étrangement négatives.

Comment remédier à ce problème si l'on souhaite par exemple connaître la factorielle de 25 ?

Eh bien le langage Ada a tout prévu. Si les Integer ne suffisent plus, il est possible d'utiliser d'autres types : **Long_Integer** (avec le package **Ada.Long_Integer_Text_IO**) et **Long_Long_Integer** (avec le package **Ada.Long_Long_Integer_Text_IO**). Les **Long_Integer** sont codés sur 4 octets soit 32 bits comme les Integer. Pourquoi ? Eh bien cela dépend de vos distributions en fait. Sur certaines plateformes, les Integer ne sont codés que sur 16 bits soit 2 octets d'où la nécessité de faire appel aux **Long_Integer**. Pour augmenter vos capacités de calcul, utilisez donc plutôt les **Long_Long_Integer** qui sont codés sur 8 octets soit 64 bits. Ils s'entendent donc de +9 223 372 036 854 775 807 à -9 223 372 036 854 775 808, ce qui nous fait 2^{64} fois plus de nombres que les **Long_Integer**.

Il existe également des types **Short_Integer** codés sur 2 octets soit 16 bits (avec le package **Ada.Short_Integer_Text_IO**) et **Short_Short_Integer** codés sur 1 octet soit 8 bits (avec le package **Ada.Short_Short_Integer_Text_IO**). Le plus grand **Short_Integer** est donc 32 767 ; le plus grand des **Short_Short_Integer** est quant à lui 127. Pratique pour de petits calculs ne nécessitant pas d'encombrer la mémoire.

Ce problème de portabilité du type Integer incitera les programmeurs expérimentés à préférer l'utilisation de types personnels plutôt que du type Integer afin d'éviter tout problème.

Utiliser les bases

En Ada, il est possible de travailler non plus avec des nombres entiers en base 10 mais avec des nombres binaires ou des nombres en base 3, 5, 8, 16... On parlera d'Octal pour la base 8 et d'hexadécimal pour la base 16 (bases très utilisées en informatique). Par exemple, nous savons que le nombre binaire 101 correspond au nombre 5. Pour cela, nous écrivons :

Code : Ada

```
N : Integer;
BEGIN
  N := 2#101#;
  Put(N);
```

Le premier nombre (2) correspond à la base. Le second (101) correspond au nombre en lui-même exprimé dans la base indiquée :

Attention ! En base 2, les chiffres ne doivent pas dépasser 1. Chacun de ces nombres doit être suivi d'un dièse (#). Le souci, c'est que ce programme affichera ceci :

Code : Console

5

Il faudra donc modifier notre code pour spécifier la base voulue : Put(N, Base => 2) ; ou Put(N, 5, 2) ; (le 5 correspondant au nombre de chiffres que l'on souhaite afficher). Votre programme affichera alors votre nombre en base 2.

Il sera possible d'écrire également :

Code : Ada

```
N := 8#27# --correspond au nombre 7+8*2 = 23
N := 16#B# --en hexadécimal, A correspond à 10 et B à 11
```

Il est même possible d'écrire un exposant à la suite de notre nombre :

Code : Ada

```
N := 2#101#E3 ;
```

Notre variable N est donc donnée en base 2, elle vaut donc 101 en binaire, soit 5. Mais l'exposant E3 signifie que l'on doit multiplier 5 par $2^3 = 8$. La variable N vaut donc en réalité 40. Ou, autre façon de voir la chose : notre variable N ne vaut pas 101 (en binaire bien sûr) mais 101000 (il faut en réalité ajouter 3 zéros).

Imposer la taille

Imaginez que vous souhaitez réaliser un programme manipulant des octets. Vous créez un type modulaire T_Octet de la manière suivante :

Code : Ada

```
Type T_Octet is mod 2**8 ;
```

Vous aurez ainsi des nombres allant de 0000 0000 jusqu'à 1111 1111. Sauf que pour les enregistrer, votre programme va tout de même réquisitionner 4 octets. Du coup, l'octet 0100 1000 sera enregistré ainsi en mémoire : 00000000 00000000 01001000. Ce qui nous fait une perte de trois octets. Minime me direz-vous. Sauf que si l'on vient à créer des tableaux d'un million de cases de types T_Octets, pour créer un nouveau format de fichier image par exemple, nous perdrons $3 \times 1000000 = 3M$ pour rien. Il est donc judicieux d'imposer certaines contraintes en utilisant l'attribut 'Size de la sorte

Code : Ada

```
Type T_Octet is mod 2**8;
For T_Octet'size use 8 ;
```

Ainsi, nos variables de type T_Octet, ne prendront réellement que 1 octet en mémoire.



L'instruction use est utilisée pour tout autre chose que les packages ! Ada est finalement très souple, derrière son apparence rigoureuse 😊



Si vous écrivez « For T_Octet'size use 5 ; », le compilateur vous expliquera que 5 bits sont insuffisants : size for "T_Octet" too small, minimum allowed is 8. Donc en cas de doute, vous pouvez toujours comptez sur GNAT.

Représentation du texte

Avant de nous attaquer à la représentation des décimaux et donc des Float, qui est particulièrement complexe, voyons rapidement celle du texte. Vous savez qu'un String n'est rien de plus qu'un tableau de caractères et qu'un Unbounded_String est en fait une liste de Characters. Mais comment sont représentés ces caractères en mémoire ? Eh bien c'est très simple, l'ordinateur n'enregistre pas des lettres mais des nombres.



Hein ? Et comment il retrouve les lettres alors ?

Grâce à une table de conversion. L'exemple le plus connu est ce que l'on appelle la table ASCII (American Standard Code for Information Interchange). Il s'agit d'une table de correspondance entre des caractères et le numéro qui leur est attribué. Cette table (elle date de 1961) est certainement la plus connue et la plus utilisée. La voici en détail sur la figure suivante.

Numerô	Symbol	Numerô	Symbol	Numerô	Symbol	Numerô	Symbol
0	NUL	32	SP (Space)	64	@	96	«
1	SOH (Start Of Heading)	33	!`	65	A	97	a
2	STX (Start Of Text)	34	,	66	B	98	b
3	ETX (End Of Text)	35	#	67	C	99	c
4	EOT (End Of Transmission)	36	\$	68	D	100	d
5	ENQ (Enquiry)	37	%	69	E	101	e
6	ACK (Acknowledge)	38	&	70	F	102	f
7	BEL (Bell)	39	,	71	G	103	g
8	BS (BackSpace)	40	(72	H	104	h
9	TAB (Horizontal Tabulation)	41)	73	I	105	i
10	LF (Line Feed, saut de ligne)	42	*	74	J	106	j
11	VT (Vertical Tabulation)	43	+	75	K	107	k
12	FF (Form Feed)	44	,	76	L	108	l
13	CR (Carriage Return, retour à la ligne)	45	-	77	M	109	m
14	SO (Shift Out)	46	=	78	N	110	n
15	SI (Shift In)	47	/	79	O	111	o
16	DLE (Data Link Escape)	48	0	80	P	112	p
17	DC1 (Device Control 1)	49	1	81	Q	113	q
18	DC2 (Device Control 2)	50	2	82	R	114	r
19	DC3 (Device Control 3)	51	3	83	S	115	s
20	DC4 (Device Control 4)	52	4	84	T	116	t
21	NAK (Negative Acknowledge)	53	5	85	U	117	u
22	SYN (Synchronous idle)	54	6	86	V	118	v
23	ETB (End of Transmission Block)	55	7	87	W	119	w
24	CAN (Character Delete)	56	8	88	X	120	x
25	EM (End of Medium)	57	9	89	Y	121	y
26	SUB (Substitute)	58	:	90	Z	122	z
27	ESC (Escape)	59	;	91	[123	{
28	FS (File Separator)	60	<	92	\	124	
29	GS (Group Separator)	61	=	93]	125	}
30	RS (Record Separator)	62	>	94	^	126	_
31	US (Unit Separator)	63	?	95	_(underscore)	127	Touche de suppression

Table ASCII Standard

Comme vous pouvez le voir, cette table contient 128 caractères allant de 'A' à 'Z' en passant par '!', '#' ou des caractères non imprimables comme la touche de suppression ou le retour à la ligne. Vous pouvez dès lors et déjà constater que seuls les caractères anglo-saxons sont présents dans cette table (pas de 'é' ou de 'é') ce qui est contrariant pour nous Français. De plus, cette table ne comporte que 128 caractères, soit 2⁷. Chaque caractère ne nécessite donc que 7 bits pour être codé en mémoire, même pas un octet ! Heureusement, cette célèbre table s'est bien développée depuis 1961, donnant naissance à de nombreuses autres tables. Le langage Ada n'utilise donc pas la table ASCII classique mais une sorte de table ASCII modifiée et étendue. En utilisant le 8ème bit, cette table permet d'ajouter 128 nouveaux caractères et donc de bénéficier de 2⁸ = 256 caractères en tout.



Donc tu nous mens depuis le début ! Il est possible d'utiliser le 'é' ! 😊

Non, je ne vous ai pas menti et vous ne pouvez pas tout à fait utiliser les caractères accentués comme les autres. Ainsi si vous écrivez ceci :

Code : Ada

```
... C : Character
BEGIN
  C := 'é' ; Put(C) ;
...
```

Vous aurez droit à ce genre d'horreur :

Code : Console

Ù

Vous devrez donc écrire les lignes ci-dessous si vous souhaitez obtenir un e minuscule avec un accent circonflexe :

Code : Ada

```
...
C : Character
BEGIN
  C := Character'val(136) ; Put(C);
...
```

Enfin, dernière remarque, les caractères imprimés dans la console ne correspondent pas toujours à ceux imprimés dans un fichier texte. Ainsi du caractère N°230 : il vaut 'à' dans un fichier texte mais 'ù' dans la console. Je vous fournis ci-dessous la liste des caractères tels qu'ils apparaissent dans un fichier texte :

0 :	2 :	3 :	4 :	5 :	6 :	7 :	8 :
1 :	10 :	11 :	12 :	13 :	14 :	15 :	16 :
9 :	18 :	19 :	20 :	21 :	22 :	23 :	24 :
17 :	26 :	27 :	28 :	29 :	30 :	31 :	32 :
25 :	34 :	35 :	36 :	37 :	38 :	39 :	40 :
33 :	42 :	43 :	44 :	45 :	46 :	47 :	48 :
41 :	50 :	51 :	52 :	53 :	54 :	55 :	56 :
49 :	58 :	59 :	60 :	61 :	62 :	63 :	64 :
57 :	65 :	66 :	67 :	68 :	69 :	70 :	71 :
65 :	73 :	74 :	75 :	76 :	77 :	78 :	79 :
73 :	81 :	82 :	83 :	84 :	85 :	86 :	87 :
81 :	89 :	90 :	91 :	92 :	93 :	94 :	95 :
97 :	98 :	99 :	100 :	101 :	102 :	103 :	104 :
105 :	106 :	107 :	108 :	109 :	110 :	111 :	112 :
113 :	114 :	115 :	116 :	117 :	118 :	119 :	120 :
121 :	122 :	123 :	124 :	125 :	126 :	127 :	128 :
129 :	130 :	131 :	132 :	133 :	134 :	135 :	136 :
137 :	138 :	139 :	140 :	141 :	142 :	143 :	144 :
145 :	146 :	147 :	148 :	149 :	150 :	151 :	152 :
153 :	154 :	155 :	156 :	157 :	158 :	159 :	160 :
161 :	162 :	163 :	164 :	165 :	166 :	167 :	168 :
169 :	170 :	171 :	172 :	173 :	174 :	175 :	176 :
177 :	178 :	179 :	180 :	181 :	182 :	183 :	184 :
185 :	186 :	187 :	188 :	189 :	190 :	191 :	192 :
193 :	194 :	195 :	196 :	197 :	198 :	199 :	200 :
201 :	202 :	203 :	204 :	205 :	206 :	207 :	208 :
209 :	210 :	211 :	212 :	213 :	214 :	215 :	216 :
217 :	218 :	219 :	220 :	221 :	222 :	223 :	224 :
225 :	226 :	227 :	228 :	229 :	230 :	231 :	232 :
233 :	234 :	235 :	236 :	237 :	238 :	239 :	240 :
241 :	242 :	243 :	244 :	245 :	246 :	247 :	248 :
249 :	250 :	251 :	252 :	253 :	254 :	255 :	256 :

Caractères après écriture dans un fichier

Et tel qu'ils apparaissent dans la console :

9 :	0 :	2 :	3 :	4 :	5 :	6 :	7 :	8 :
9 :	14 :	15 :	16 :	17 :	18 :	19 :	20 :	21 :
25 :	26 :	27 :	28 :	29 :	30 :	31 :	32 :	33 :
34 :	35 :	36 :	37 :	38 :	39 :	40 :	41 :	42 :
43 :	44 :	45 :	46 :	47 :	48 :	49 :	50 :	51 :
52 :	53 :	54 :	55 :	56 :	57 :	58 :	59 :	60 :
61 :	62 :	63 :	64 :	65 :	66 :	67 :	68 :	69 :
70 :	71 :	72 :	73 :	74 :	75 :	76 :	77 :	78 :
79 :	80 :	81 :	82 :	83 :	84 :	85 :	86 :	87 :
88 :	89 :	90 :	91 :	92 :	93 :	94 :	95 :	96 :
97 :	98 :	99 :	100 :	101 :	102 :	103 :	104 :	105 :
106 :	107 :	108 :	109 :	110 :	111 :	112 :	113 :	114 :
115 :	116 :	117 :	118 :	119 :	120 :	121 :	122 :	123 :
124 :	125 :	126 :	127 :	128 :	129 :	130 :	131 :	132 :
133 :	134 :	135 :	136 :	137 :	138 :	139 :	140 :	141 :
142 :	143 :	144 :	145 :	146 :	147 :	148 :	149 :	150 :
151 :	152 :	153 :	154 :	155 :	156 :	157 :	158 :	159 :
160 :	161 :	162 :	163 :	164 :	165 :	166 :	167 :	168 :
169 :	170 :	171 :	172 :	173 :	174 :	175 :	176 :	177 :
178 :	179 :	180 :	181 :	182 :	183 :	184 :	185 :	186 :
187 :	188 :	189 :	190 :	191 :	192 :	193 :	194 :	195 :
196 :	197 :	198 :	199 :	200 :	201 :	202 :	203 :	204 :
205 :	206 :	207 :	208 :	209 :	210 :	211 :	212 :	213 :
214 :	215 :	216 :	217 :	218 :	219 :	220 :	221 :	222 :
223 :	224 :	225 :	226 :	227 :	228 :	229 :	230 :	231 :
232 :	233 :	234 :	235 :	236 :	237 :	238 :	239 :	240 :
241 :	242 :	243 :	244 :	245 :	246 :	247 :	248 :	249 :
250 :	251 :	252 :	253 :	254 :	255 :	256 :	257 :	258 :

Caractères après écriture dans le terminal.

Vous remarquerez les ressemblances avec la table ASCII, mais aussi les différences quant aux caractères spéciaux. Le caractère n°12 correspond au saut de page et le n°9 à la tabulation.

Représentation des nombres décimaux en virgule flottante

Nous arrivons enfin à la partie la plus complexe. Si l'on parait encore compliqué de passer du binaire au décimal avec les nombres entiers, je vous conseille de relire la sous-partie concernée avant d'attaquer celle-ci car la représentation des float est un peu tirée par les cheveux. Et vous devrez avoir une notion des puissances de 10 pour comprendre ce qui suit (je ne ferai pas de rappels, il ne s'agit pas d'un cours de Mathématiques).

Représentation des float

Représentation en base 10

Avant toute chose, nous allons voir comment écrire différemment le nombre **-9876,0123** dans notre base 10. Une convention, appelée écriture scientifique, consiste à n'écrire qu'un seul chiffre, différent de 0, avant la virgule : **9,8760123**. Mais pour comprendre cette modification, il faut multiplier le résultat obtenu par 1000 pour indiquer qu'en réalité il faut décaler la virgule de trois chiffres vers la droite. Ainsi, **-9876,0123** s'écrit **-9,8760123 × 10³** ou même **-9,8760123E3**.



Eh ! Ça me rappelle les écritures bizarres qu'on obtient quand on utilise Put() avec des float !

Eh oui, par défaut, c'est ainsi que sont représentés les nombres décimaux par l'ordinateur, le E3 signifiant $\times 10^3$. Il suffit donc simplement de modifier l'espacement (le E3) pour décaler la virgule vers la gauche ou la droite : on parle ainsi de **virgule flottante**.

Représentation en binaire

Cette écriture en virgule flottante prend tout son sens en binaire : pour enregistrer le nombre **-1001,1** il faudra enregistrer en réalité le nombre **-1,001E3**. Mais comme le chiffre avant la virgule ne peut pas être égal à 0, ce ne peut être qu'un 1 en binaire ! Autrement dit, il faudra enregistrer trois nombres :

- le **signe** : 1 (le nombre est négatif)
- le **exposant** : 11 (pour E3, rappelons que 3 s'écrit 11 en binaire)
- la **mantisso** : 0011 (on ne conserve pas le 1 avant la virgule, c'est inutile)

D'où une écriture en mémoire qui ressemblerait à cela :

Signe	Exposant	Mantisse
1	11	0011



Pourquoi ne pas avoir enregistré seulement deux nombres ? La partie entière et la partie décimale ?

Cette représentation existe, c'est la représentation dite en **virgule fixe**. Mais cette représentation rigide ne permet pas de représenter autant de nombres que la représentation en virgule flottante, bien plus souple. En effet, avec 4 octets, le nombre binaire **00000000000000000000000000000001** ne pourra pas être représenté en virgule fixe, le 1 étant situé « trop loin ». En revanche, cela ne pose aucun souci en virgule flottante, puisqu'il suffira de l'écrire ainsi : **1,0E(-17)**.

En simple précision, les nombres à virgule flottante sont représentés sur 4 octets. Comment se répartissent les différents bits ? C'est simple.

- D'abord 1 bit pour le signe,
- puis 8 bits pour l'exposant,
- enfin le reste (soit 23 bits) pour la mantisse.

Sauf que l'exposant, nous venons de le voir peut être négatif et qu'il ne va pas être simple de gérer des exposants négatifs comme nous le faisons avec les Integer. Pas question d'avoir un bit pour le signe de l'exposant et 7 pour sa valeur absolue ! Le codage est un peu plus compliqué : on va décaler l'exposant pour aller de -127 à +128. Si l'exposant vaut 0000 0000, cela correspond au plus petit des exposants : -127. Il y a un décalage de 127 par rapport à l'encodage normal des Integer.

Un exemple

Nous voulons convertir le nombre 6,75 en binaire et en virgule flottante. Tout d'abord, convertissons-le en binaire :

$$6,75 = (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})$$

Donc notre nombre s'écrit 110,11. Passons en écriture avec virgule flottante : 1,1011 $\times 2^2$ ou 1,1011E2. Rappelons qu'il faut décaler notre exposant de 2 : 2 + 127 = 129. Et comme 129 s'écrit 10000001 en binaire, cela nous donnera :

Signe	Exposant	Mantisse
0	1000 0001	1011000 0000000 0000000

Cas particuliers

Ton raisonnement est bien gentil, mais le nombre 0 n'a aucun chiffre différent de 0 avant la virgule !

En effet, ce raisonnement n'est valable que pour des nombres dont l'exposant n'est ni 0000 0000 ni 1111 1111, c'est à dire si les nombres sont normalisés. Dans le cas contraire, on associe des valeurs particulières :

- Exposant 0 Mantisse 0 : le résultat est 0.
- Exposant 0 : on tombe dans le cas des nombres dénormalisés, c'est à dire dont le chiffre avant la virgule est 0 (nous n'entrons pas davantage dans le détail pour ce cours).
- Exposant 1111 1111 Mantisse 0 : le résultat est l'infini.
- Exposant 1111 1111 : le résultat n'est pas un nombre (Not A Number : NaN).

Remarquez que du coup, pour les flottants, il est possible d'avoir +0 et -0 ! Bien. Vous connaissez maintenant les bases du codage des nombres flottants (pour plus de précisions, voir le [tutoriel de Mewtwo](#)), il est grand temps de voir les implications que cela peut avoir en Ada.

Implications sur le langage Ada

De nouveaux types pour gagner en précision

Bien entendu, vous devez vous en douter, il existe un type Short_Float (mais pas short_short_float), et il est possible de travailler avec des nombres à virgule flottante de plus grande précision à l'aide des types Long_Float (codé sur 8 octets au lieu de 4, c'est le format double précision) et Long_Long_Float (codé sur 16 octets). Vous serez alors amenés à utiliser les packages Ada.Short_Float_IO, Ada.Long_Float_IO et Ada.Long_Long_Float_IO (mais ai-je encore besoin de le spécifier ?).

Problèmes liés aux arrondis

Les nombres flottants posent toutefois quelques soucis techniques. En effet, prenons un nombre de type float. Il dispose de 23 bits pour coder sa mantisse, plus un «bit fantôme» puisque le chiffre avant la virgule n'a pas besoin d'être codé. Donc il est possible de coder un nombre de 24 chiffres, mais pas de 25 chiffres. Par exemple $2^{25} + 1$ en binaire, correspond à un nombre de 26 chiffres : le premier et le dernier vaudront 1, les autres 0. Essayons le code suivant :

Code : Ada

```
...
x : Float;
Begin
  x := 2.0**25;
  put(Integer(x), base => 2); new_line; --on écriture en binaire (c'est un entier)
  put(x, exp => 0); new_line; --et on affiche le nombre flottant
  x := x + 1.0; --on incrémente
  put(Integer(x), base => 2); new_line; --et reboucle pour l'affichage
  put(x, exp => 0);
...
```

Que nous renvoie le programme ? Deux fois le même résultat ! Étrange. Pourtant nous avons bien incrémenté notre variable x ! Je vous ai déjà donné la réponse dans le précédent paragraphe : le nombre de bits nécessaire pour enregistrer les chiffres (on parle de chiffres significatifs) est limité à 23 (en comptant le bit avant la virgule). Si nous dépassons ces capacités, il ne se produit pas de plantage comme avec les Integer, mais le processeur renvoie un résultat arrondi d'où une perte de précision.

De même, si vous testez le code ci-dessous, vous devriez avoir une drôle de surprise. Petite explication sur ce code : on effectue une boucle avec deux compteurs. L'un est un flottant que l'on augmente de 0.1 à chaque itération, l'autre est un integer que l'on incrémente. Nous devrions donc avoir 1001 itérations et à la fin, n vaudra 1001 et x vaudra 100.1. Logique, non ? 😊 Eh bien testez donc ce code.

Code : Ada

```
...
x := 0.0;
n := 0;
while x < 100.0 loop
  x := x + 0.1;
  n := n + 1;
end loop;
Put("x vaut "); put(x,exp=>0); new_line;
Put_line("n vaut" & Integer'image(n));
```

Alors, que vaut n ? 1001, tout va bien. Mais que vaut x ? 100.09904 !!! 😊 Qu'a-t-il bien pu se passer encore ? Pour comprendre, il faut savoir que le nombre décimal 0.1 s'écrit approximativement 0.000 100 1000 1100 1100 en binaire... Cette écriture est infinie, donc l'ordinateur doit effectuer un arrondi. Si la conversion d'un nombre entier en binaire tombe toujours juste, la conversion d'un nombre décimal quant à elle peut poser problème car l'écriture peut être potentiellement infinie. Et même chose lorsque l'on transforme le nombre binaire en nombre décimal, on ne retrouve pas 0.1 mais une valeur très proche ! Cela ne pose pas de véritable souci lorsque l'on se contente d'effectuer une addition. Mais lorsque l'on effectue de nombreuses additions, ce problème d'arrondi commence à se voir. La puissance des ordinateurs et la taille du type float (4 octets) est largement suffisante pour pallier dans la plupart des cas à ce problème d'arrondi en fournissant une précision bien supérieure à nos besoins, mais cela peut toutefois être problématique dans des cas où la précision est de mise.

Car si cette perte de précision peut paraître minime, elle peut parfois avoir de lourdes conséquences : imaginez que votre programme calcule la vitesse qu'une fusée doive adopter pour se poser sans encombre sur Mars en effectuant pour cela des milliers de boucles ou qu'il serve à la comptabilité d'une grande banque générant des millions d'opérations à la seconde. Une légère erreur d'arrondi qui viendrait à se répéter des milliers ou des millions de fois pourrait à terme engendrer la destruction de votre fusée ou une crise financière mondiale (comment ça ? Vous trouvez que j'exagère ? 😊).

Plus simplement, ces problèmes d'arrondis doivent vous amener à une conclusion : les types flottants ne doivent pas être utilisés à la légère ! Partout où les types entiers sont utilisables, préferez-les aux flottants (vous comprenez maintenant pourquoi depuis le début, je recharge à utiliser le type Float). N'utilisez pas de flottant comme compteur dans une boucle car les problèmes d'arrondis pourraient bien vous jouer des tours.

Delta et Digits

Toujours pour découvrir les limites du type float, testez le code ci-dessous :

Code : Ada

```
...  
x : float;  
begin  
x:=12.3456789;  
put(x,exp=>0);  
...
```

Votre programme devrait afficher la valeur de x, or il n'affiche que :

Code : Console

```
12.34568
```

Eh oui, votre type float ne prend pas tous les chiffres ! Il peut manipuler de très grands ou de très petits nombres, mais ceux-ci ne doivent pas, en base 10, excéder 7 chiffres significatifs, sinon ... il arrondit ! Ce n'est pas très grave, mais cela constitue encore une fois une perte de précision. Alors certes vous pourriez utiliser le type Long_Float qui gère 16 chiffres significatifs, ou long_long_Float qui en gère 20 ! Mais il vous est également possible en Ada de définir votre propre type flottant ainsi que sa précision. Exemple :

Code : Ada

```
type MyFloat is digits 9;  
x : MyFloat := 12.3456789;
```

Le mot **digits** indiquera que ce type MyFloat générera au moins 9 chiffres significatifs ! Pourquoi "au moins" ? Eh bien parce qu'en créant ainsi un type Myfloat, nous créons un type de nombre flottant particulier utilisant un nombre d'octet particulier. Ici, notre type Myfloat sera codé dans votre ordinateur de sorte que sa mantisse utilise 53 bits. Ce type Myfloat pourra en fait gérer des nombres ayant jusqu'à 15 chiffres significatifs, mais pas au-delà.



Pour savoir le nombre de bits utilisés pour la mantisse de votre type, vous n'aurez qu'à écrire :

```
Put(Myfloat'Machine_Mantissa);
```

Enfin, nous avons beaucoup parlé des nombres en virgule flottante, car c'est le format le plus utilisé pour représenter les nombres décimaux. Mais comme évoqué plus haut, il existe également des nombres en virgule fixe. L'idée est simple : le nombre de chiffres après la virgule est fixé dès le départ et ne bouge plus. Inconvénient : si vous avez fixé à 2 chiffres après la virgule, vous pourrez oublier les décimaux avec 0.00000 ! Mais ces nombres ont l'avantage de permettre des calculs plus rapides pour l'ordinateur. Vous en avez déjà rencontré : les types Duration ou Time sont des types à virgule fixe (donc non flottante). Pour créer un type de nombre en virgule fixe vous procédez ainsi :

Code : Ada

```
type T_Prix is delta 0.01;  
Prix_CD : T_Prix := 15.99;
```

Le mot clé **delta** indique l'écart minimal pouvant séparer deux nombres en virgule fixe et donc le nombre de chiffres après la virgule (2 pas plus) : ici, notre prix de 15€99, s'il devait augmenter, passerait à 15,99 + 0,01 = 16€. Une précision toutefois : il est nécessaire de combiner **delta** avec **digits** ou **range** ! Exemple :

Code : Ada

```
type T_Livret_A is delta 0.01 digits 6;
```

Toute variable de type T_Livret_A aura deux chiffres après la virgule et 6 chiffres en tout. Sa valeur maximale sera donc 9999,99. Une autre rédaction serait :

Code : Ada

```
type T_Livret_A is delta 0.01 range 0.0 .. 9999.99;
```

Toutefois, gardez à l'esprit que malgré leurs limites (arrondis, overflow ...) les nombres en virgule flottante demeurent bien plus souples que les nombres en virgule fixe. Ces derniers ne seront utilisés que dans des cas précis (temps, argent ...) où le nombre de chiffres après la virgule est connu à l'avance.

En résumé :

- L'encodage du type **Integer** peut dépendre de la machine utilisée. C'est pourquoi il est parfois utile de définir son propre type entier ou de faire appel au type **Long_Long_Integer** afin de vous mettre à l'abri des problèmes d'overflow.
- Le type **Character** gère 256 symboles numérotés de 0 à 255, dont certains ne sont pas imprimables. Les caractères latins sont accessibles via leur position, en utilisant les attributs **character'pos()** et **character'val()**.
- Le type **Float** et autres types à virgule flottante, offrent une grande souplesse dans l'utilisation des nombres décimaux. Toutefois, ils sont soumis à des problèmes d'arrondis qui peuvent s'accumuler au fil des boucles récursives ou itératives.
- Les nombres à virgule fixe sont plus faciles à comprendre mais impliquent beaucoup de restrictions. Réservez-les à quelques types spécifiques dont vous connaissez par avance le nombre maximal de chiffres avant et après la virgule.

La programmation modulaire II : Encapsulation

Avec ce chapitre, nous allons entrer de plein pied dans la Programmation Orientée Objet (souvent appelée par son doux acronyme de POO). L'objectif de ce chapitre est double :

- expliquer ce que sont les notions de POO, de classes...;
- fournir une première application de ce concept de POO en abordant l'encapsulation.

Cette notion de Programmation Orientée Objet est une notion compliquée que nous allons prendre le temps d'aborder au travers de ce chapitre et des quatre suivants (eh oui, cinq chapitres, ce n'est pas de trop). Alors, prenez un grand bol d'air car avec la POO nous allons faire une longue plongée dans une nouvelle façon de programmer de l'utiliser nos packages, en commençant par l'encapsulation.

Qu'est-ce qu'un objet ?

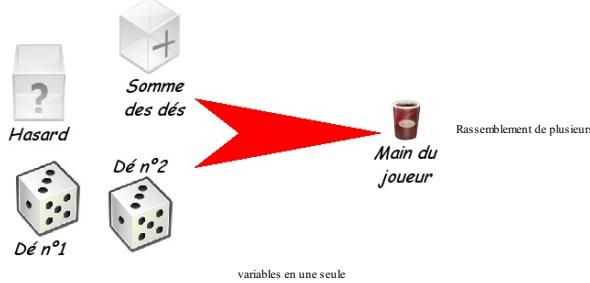
Une première approche

Au cours de la partie III, nous avons manipulé des types plus complexes : tableaux, pointeurs, types structurés... nous nous sommes ainsi approchés de cette notion d'objet en manipulant ce qui n'était déjà plus de simples variables. Je parlais ainsi de variables composites, mais il ne s'agissait pas encore de véritables objets.



Mais alors, c'est quoi un objet ? Si c'est encore plus compliqué que les pointeurs et les types structurés j'abandonne !

Vous vous souvenez de l'époque où nous avions réalisé notre TP sur le craps ? À la suite de cela, je vous avais dit qu'il aurait été plus judicieux que nos nombreuses variables soient toutes rassemblées en un seul et même contenant :

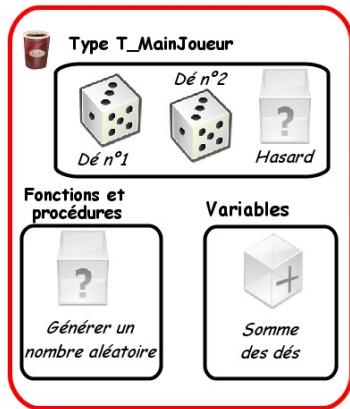


Avec nos connaissances actuelles, nous savons que ces différentes variables pourraient être rassemblées en un seul type structuré de la manière suivante :

Code : Ada

```
type T_MainJoueur is record
    de1, de2 : natural range 1..6;
    hasard : generator;
    somme : natural range 2..12;
end record;
```

Voire mieux encore, nous pourrions tout rassembler (le type T_MainJoueur, les variables, les fonctions et les procédures associées) dans un seul package appelé P_MainJoueur ! À la manière suivante :



Rassemblement en un seul package

Ainsi, nous aurions au sein d'un même paquetage à la fois les variables, les types et les programmes nécessaires à son utilisation (génération des dés, lecture de leur somme...). Bref, aujourd'hui, vous ne procéderiez plus comme à l'époque où tout était en vrac dans le programme principal.



En effet, j'admette avoir fait d'énormes progrès en très peu de temps, en primaire déjà mes instituteurs... Eh ! Mais tu dérives là ! Tu ne réponds pas à ma question : C'EST QUOI UN OBJET ??!

Un objet ? C'est justement cela : un type de donnée muni d'une structure comprenant l'ensemble des outils nécessaires à la représentation d'un objet réel ou d'une notion. Vous ne devez plus voir vos packages simplement comme une bibliothèque permettant de stocker « tout et n'importe quoi du moment que ça ne me gêne plus dans mon programme principal » mais plutôt comme une entité à part entière, porteuse d'un sens et d'une logique propre. Ainsi, pour constituer un objet correspondant à la main d'un joueur, notre package P_MainJoueur devrait contenir un type T_Main_Joueur englobant les dés et tous les paramètres nécessaires) ainsi que les outils pour lire ou modifier cette main... mais rien qui ne concerne son score par exemple !

Ainsi, concevoir un objet c'est non seulement concevoir un type particulier mais également tous les outils nécessaires à sa manipulation par un tiers utilisateur et une structure pour englober tout cela.

J'insiste sur le mot « nécessaires » ! Prenez par exemple un programmeur qui a conçu un type T_Fenetres avec tout plein de procédures et de fonctions pour faire une jolie fenêtre sous Windows, Linux ou Mac. Il souhaite faire partager son travail au plus grand nombre : il crée donc un package P_Fenetres qu'il va diffuser. Ce package P_Fenetres n'a pas besoin de contenir des types T_Fenetre_3D, ce sera hors sujet. Mais évidemment, il a besoin d'accéder aux procédures Creer_Barre, De_Tire, Creer_Bouton, Fenetre, Creer_Bouton_Agrandir... ? Non, il n'a besoin que d'une procédure Creer_Fenetres : la plupart des petits programmes créés par le programmeur n'intéresseront pas l'utilisateur final : il faudra donc permettre facilement à l'utilisateur à certaines fonctionnalités, mais pas à toutes !



Mathieu Nebra a écrit sur ce sujet dans son cours sur le C++. L'illustration qu'il donne est très parlante et peut vous fournir une seconde explication si vous avez encore du mal à comprendre de quoi il retourne. Cliquez [ici](#) pour suivre son explication (partie 1 seulement).

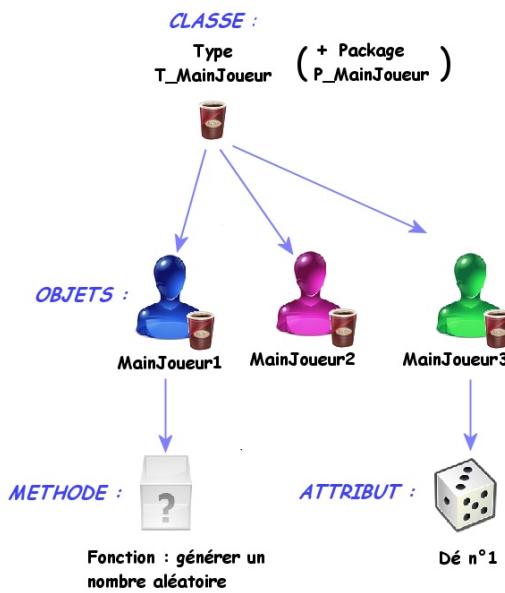
Posons le vocabulaire

Pour faire de la programmation orientée objet, il faut que nous établissons un vocabulaire précis. Le premier point à éclaircir est la différence entre les termes « classe » et « objet ». La classe correspond à notre type (nous nuancerons cette assertion plus tard) : elle doit toutefois être munie d'un package dans lequel sont décrits le type et les fonctions et procédures pour le manipuler. À partir de ce « schéma », il est possible de créer plusieurs « variables » que l'on nommera objets : on dit alors que l'on instancie des objets à partir d'une classe (de la même manière que l'on déclare des variables d'un certain type).



Cette notion de classe est encore incomplète. Elle sera approfondie lors des chapitres suivants.

Autre point de vocabulaire, les procédures et fonctions liées à une classe et s'appliquant à notre objet sont appelées **méthodes**. Pour mettre les choses au clair, reprenons l'exemple de notre jeu de craps en image :



Enfin, certaines catégories de méthodes portent un nom spécifique selon le rôle qu'elles jouent : **constructeur** pour initialiser les objets, **destructeur** pour les détruire proprement (en libérant la mémoire dynamique par exemple, nous reviendrons cela dans le chapitre sur la finalisation), **accesseur**, **modifieur** pour lire ou modifier un attribut, **itérateur** pour appliquer une modification à tous les éléments d'une pile par exemple ...

De nouvelles contraintes

On attache à la Programmation Orientée Objet, non seulement la notion de classe englobant à la fois le type et les programmes associés, mais également divers principes comme :

- L'**encapsulation** : c'est l'une des **propriétés fondamentales** de la POO. Les informations contenues dans l'objet ne doivent pas être «**obdouillables**» par utilisateur. Elles doivent être protégées et manipulées uniquement à l'aide des méthodes fournies par le package. Si l'on prend l'exemple d'une voiture : le constructeur de la voiture fait en sorte que le conducteur n'a pas besoin d'être mécanicien pour l'utiliser. Il crée un volant, des sièges et des pédales pour éviter que le conducteur ne vienne mettre son nez dans le moteur (ce qui serait un risque). Nous avons commencé à voir cette notion avec la **séparation** des spécifications et du corps des packages et nous allons y ajouter la touche finale au cours de ce chapitre avec la **privatisation**.
- La **généricité** : nous l'avons régulièrement évoquée au cours de ce tutoriel. Elle consiste à réaliser des packages applicables à divers types de données. Il en était ainsi pour les Doubly Linked Lists ou les Vectors qui pouvaient contenir des float, des integer ou des types personnalisés. Nous verrons cette notion plus en profondeur lors du prochain chapitre sur la programmation modulaire.
- L'**héritage** et la **dérivation** : deuxième notion fondamentale de la conception orientée objet. Plus complexe, elle sera traitée après la généricité.

Un package... privé

Partie publique / partie privée

" Ici c'est un club privé, on n'entre pas ! "

Vous avez peut-être déjà entendu cette tirade ? Eh bien nous allons pouvoir nous venger aujourd'hui en faisant la même chose avec nos packages. Bon certes, personne ne comptait vraiment venir danser avec nous devant le tuto... Disons qu'il s'agit de rendre une partie du code «inaccessible». Petite explication : lorsque nous avons rédigé nos packages P_Pile et P_File, je vous avais bien précisé que nous voulions proposer à l'utilisateur ou au programmeur final toute une gamme d'outils qui lui permettraient de ne pas avoir à se soucier de la structure des types T_File et T_Pile. Ainsi, pour accéder au deuxième élément d'une pile, il lui suffit d'écrire :

Code : Ada

```
...
n : integer ;
P : T_Pile
BEGIN
  ...
  --création de la pile
  pop(P,n) ;
  put(first(P)) ;
```

La logique est simple : il dépile le premier élément puis affiche le premier élément de la sous-pile restante. Simple et efficace ! C'est d'ailleurs pour cela que nous avions créé toutes ces primitives. Mais en réalité, rien n'empêche notre programmeur d'écrire le code suivant.

Code : Ada

```
...
P : T_Pile
BEGIN
  ...
  --création de la pile
  put(P.suivant.valeur) ;
```

Et là, c'est plutôt embêtant non ? Car nous avons rédigé des procédures et des fonctions justement dans le but qu'il ne puisse pas utiliser l'écriture avec pointeurs. Cette écriture présente des risques, nous le savons et avons pu le constater en testant nos packages P_Pile et P_File ! Alors il est hors de question que nous ayons fait tous ces efforts pour fournir un code sûr et qu'un oubli s'amuse finalement à «détourner» notre travail. Si cela n'a pas de grande incidence à notre échelle, cela peut poser des problèmes de stabilité et de sécurité du code dans de plus grands projets nécessitant la coopération de nombreuses personnes.

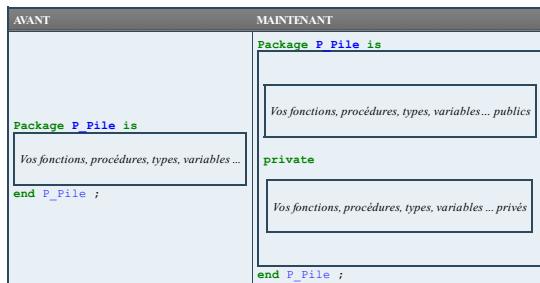


Bah oui mais on ne peut pas empêcher les gens de faire n'importe quoi !

Eh bien si, justement ! La solution consiste à rendre la structure de nos types T_Pile, T_File et T_Cellule illisible en dehors de leurs propres packages. Actuellement, n'importe qui peut la manipuler à sa guise simplement en faisant appel au bon package. On dit que nos types sont **publics**. Par défaut, tout type, variable, sous-programme... déclaré dans un package est public.

Pour restreindre l'accès à un type, une variable..., il faut donc le déclarer comme **privé** (PRIVATE en Ada). Pour réaliser cela, vous allez devoir changer votre façon de voir vos packages ou tout du moins les spécifications. Vous les voyez jusqu'alors comme une sorte de grande boîte où l'on stockait tout, n'importe quoi et n'importe comment. Vous allez devoir apprendre que

cette grande boîte est en fait compartimentée.



Comme l'indique le schéma ci-dessus, le premier compartiment, celui que vous utilisez, correspond à la partie publique du package. Il est introduit par le mot **IS**. Le second compartiment correspond à la partie privée et est introduit par le mot **PRIVATE**, mais comme cette partie était jusqu'alors vide, nous n'avions pas à la renseigner. Reprenons le code du fichier **P_Pile.ads** pour lui ajouter une partie privée :

Code : Ada

```

PACKAGE P_Pile IS
    -- PARTIE PUBLIQUE : NOS PRIMITIVES
    PROCEDURE Push (P : IN OUT T_Pile; N : IN Integer);
    PROCEDURE Pop (P : IN OUT T_Pile; N : OUT Integer);
    FUNCTION Empty (P : IN T_Pile) RETURN Boolean;
    FUNCTION Length(P : T_Pile) RETURN Integer;
    FUNCTION First(P : T_Pile) RETURN Integer;

PRIVATE
    -- PARTIE PRIVÉE : NOS TYPES
    TYPE T_Cellule;
    TYPE T_Pile IS ACCESS ALL T_Cellule;
    TYPE T_Cellule IS
        RECORD
            Valeur : Integer;
            Index : Integer;
            Suivant : T_Pile;
        END RECORD;
END P_Pile;

```

Les utilisateurs doivent garder l'accès aux primitives, elles ont été faites pour eux tout de même. En revanche, ils n'ont pas à avoir accès à la structure de nos types, donc nous les rendons privés. Et voilà ! Le tour est joué ! Pas besoin de modifier le corps de notre package, tout se fait dans les spécifications.



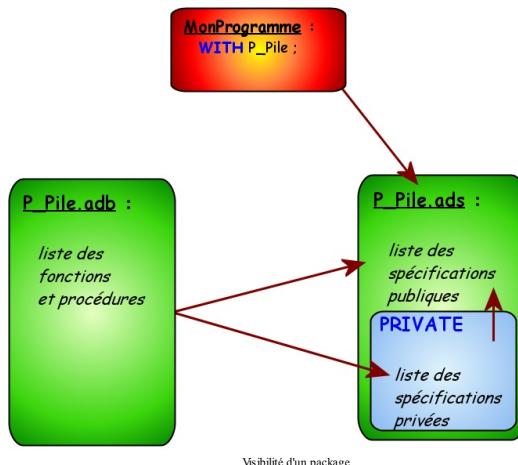
À noter qu'il est possible de déclarer des fonctions ou procédures dans la partie **PRIVATE**. Ces programmes ne pourront dès lors être utilisés que par les programmes du corps du package. Plus d'explications seront fournies ci-après.

Visibilité



Mais ça marche pas tes affaires ! 😊 À la compilation, GNAT m'indique "T_Pile" is undefined (more references follow).

Ah oui... c'est embêtant. Cela n'arrive à vous parler de la visibilité du code. Lorsque vous créez un programme faisant appel à **P_Pile** (avec instruction « **WITH P_Pile ;** »), ce programme n'a pas accès à l'intégralité du package ! Il aura simplement accès aux spécifications et, plus précisément encore, aux spécifications publiques ! Toute la partie privée et le corps du package demeurent invisibles, comme l'indique le schéma suivant.



Une flèche indique qu'une partie « a accès à » ce qui est écrit dans une autre ; On voit ainsi que seul le corps du package sait comment est constituée la partie privée puisqu'il a une pleine visibilité sur l'ensemble du fichier **ads**. Donc seul le **BODY** peut manipuler librement ce qui est dans la partie **PRIVATE**.



Comment peut-on alors créer un type **T_Pile** si ce type est invisible ?

L'utilisateur doit pouvoir voir ce type sans pour autant pouvoir le lire. « *Voir notre type* » signifie que l'utilisateur peut déclarer des objets de type **T_Pile** et utiliser des programmes utilisant eux-mêmes ce type. En revanche, il ne doit pas en connaître la structure, c'est ce que l'appelle ne pas pouvoir « *lire notre type* ». Comment faire cela ? Eh bien, nous devons indiquer dans la partie publique que nous disposons de types privés. Voici donc le code du package **P_Pile** tel qu'il doit être écrit (concentrez-vous sur la ligne 5) :

Code : Ada

```

PACKAGE P_Pile IS
    -- PARTIE PUBLIQUE
    TYPE T_Pile IS PRIVATE; --On indique qu'il y a un type T_Pile
    --mais que celui-ci est privé
    PROCEDURE Push (P : IN OUT T_Pile; N : IN Integer);
    PROCEDURE Pop (P : IN OUT T_Pile; N : OUT Integer);
    FUNCTION Empty (P : IN T_Pile) RETURN Boolean;
    FUNCTION Length(P : T_Pile) RETURN Integer;
    FUNCTION First(P : T_Pile) RETURN Integer;

PRIVATE

```

```
-- PARTIE PRIVÉE
TYPE T_Cellule;
TYPE T_Pile IS ACCESS ALL T_Cellule; --on décrit ce qu'est
le type T_Pile
TYPE T_Cellule IS
  RECORD
    Valeur : Integer;
    Index : Integer;
    Suivant : T_Pile;
  END RECORD;
END P_Pile;
```

Et cette fois, c'est fini : votre type T_Pile est visible mais plus lisible, vous avez créé un véritable type privé en encapsulant le type T_Cellule.

Un package privé et limité

Que faire avec un type PRIVATE ?

Nous avons dit que le type T_Pile était privé et donc qu'il était visible mais non lisible. Par conséquent, il est impossible de connaître la structure d'une pile P et donc d'écrire :

Code : Ada - CODE FAUX

```
P.valeur := 15;
```

En revanche, il est possible de déclarer des variables de type T_Pile, d'affecter une pile à une autre ou encore de tester l'égalité ou l'inégalité de deux piles :

Code : Ada

```
-- TOUTES LES OPÉRATIONS CI-DESSOUS SONT THÉORIQUEMENT POSSIBLES
P1, P2, P3 : T_Pile           --déclaration
BEGIN
  ...
  P2 := P1;                  --affectation
  if P2 = P1 and P3/=P1     --comparaison (égalité et inégalité
  seulement)
  then ...
```

Mais à moins que vous ne les surchargez, les opérateurs +, *, / ou - ne sont pas définis pour nos types T_Pile et ne sont donc pas utilisables, de même pour les tests d'infériorité (< et <=) ou de supériorité (> et >=).

Code : Ada - CODE FAUX

```
P3 := P1 + P2;
if P3 < P1
then ...
```

Restreindre encore notre type

Type limité et privé

Eh ! Mais si P1, P2 et P3 sont des piles, ce sont donc des pointeurs ! C'est pas un peu dangereux d'écrire P2 := P1 dans ce cas ?

Si, en effet. Vous avez bien retenu la leçon sur les pointeurs. Il est effectivement risqué de laisser à l'utilisateur la possibilité d'effectuer des comparaisons et surtout des affectations. Heureusement, le langage Ada a tout prévu ! Pour restreindre encore les possibilités, il est possible de créer un type non plus **PRIVATE**, mais **LIMITED PRIVATE** (privé et limité) ! Et c'est extrêmement facile à réaliser puisqu'il suffit juste de modifier la ligne n°5 du fichier ads :

Code : Ada

```
PACKAGE P_Pile IS
  -- PARTIE PUBLIQUE
  TYPE T_Pile IS LIMITED PRIVATE; --Le type T_Pile est désormais
  privé ET limité ! !
  PROCEDURE Push (P : IN OUT T_Pile; N : IN Integer);
  PROCEDURE Pop (P : IN OUT T_Pile; N : OUT Integer);
  FUNCTION Empty (P : IN T_Pile) RETURN Boolean;
  FUNCTION Length(P : T_Pile) RETURN Integer;
  FUNCTION First(P : T_Pile) RETURN Integer;

  PRIVATE
    -- PARTIE PRIVÉE
    TYPE T_Cellule;
    TYPE T_Pile IS ACCESS ALL T_Cellule; --on décrit ce qu'est
    le type T_Pile
    TYPE T_Cellule IS
      RECORD
        Valeur : Integer;
        Index : Integer;
        Suivant : T_Pile;
      END RECORD;
    END P_Pile;
```

Désormais, l'utilisateur ne pourra QUE déclarer ses piles ! Plus d'affectation ni de comparaison !

Mais ??! À quoi ça sert d'avoir un objet si on ne peut pas lui affecter de valeur ?

Pensez aux fichiers et au type File_Type. Avez-vous besoin d'effectuer des affectations ou des comparaisons ? Non bien sûr. Cela n'aurait eu aucun sens. Mais cela ne vous a pas empêché d'employer des objets de type File_Type et de leur appliquer diverses méthodes comme Put(), Get(), End_Of_File() et tant d'autres. Le principe est exactement le même dans le cas présent : est-ce cohérent d'affecter UNE valeur à une pile ? Je ne crois pas. D'ailleurs, nous avons la primitive Push() pour ajouter une valeur à celles déjà existantes. En revanche, il pourrait être utile d'ajouter quelques méthodes à notre package P_Pile pour copier une pile dans une autre ou pour tester si les piles contiennent un ou des éléments identiques. Le but étant d'avoir un contrôle total sur les affectations et les tests : avec un type **LIMITED PRIVATE**, rien à part la déclaration ne peut se faire sans les outils du package associé.

Type seulement limité

Pour information, il est possible de déclarer un type qui soit seulement **LIMITED**. Par exemple :

Code : Ada

```
TYPE T_Vainqueur IS LIMITED RECORD
  Nom : String(1..3);
  Score : Natural := 0;
END RECORD;
```

Ainsi, il sera impossible d'effectuer une affectation directe ou un test d'égalité (ou inégalité) :

Code : Ada - CODE FAUX

```
... Moi,Toi : T_Vainqueur;
BEGIN
```



```
Toi := ("BOB",5300) ;      -- Erreurs ! ! ! Pas
d'affectation possible !
Toi := Toi
if Moi = Toi           -- Deuxième erreur ! ! ! Pas de
comparaison possible !
then ...
```

En revanche, puisque le type T_Vainqueur n'est pas privé, sa structure est accessible et il demeure possible d'effectuer des tests ou des affectations sur ses attributs. Le code suivant est donc correct.

Code : Ada

```
Toi.nom := "JIM";
if Moi.score > Toi.score
  then put("Je t'ai mis la misere ! ! ! ");
end if;
```

L'intérêt d'un type uniquement **LIMITED** est plus discutable et vous ne le verrez que très rarement. En revanche, les types **LIMITED PRIVATE** vont devenir de plus en plus courants, croyez-moi.

Exercices

Exercice 1

Énoncé

Sur le même principe, modifier votre package P_File afin que votre type T_File soit privé et limité.

Solution

Secret (cliquez pour afficher)

Code : Ada

```
PACKAGE P_Pile IS
  TYPE T_Pile IS LIMITED PRIVATE ;
  PROCEDURE Push (P : IN OUT T_Pile; N : IN Integer) ;
  PROCEDURE Pop (P : IN OUT T_Pile; N : OUT Integer) ;
  FUNCTION Empty (P : IN T_Pile) RETURN Boolean ;
  FUNCTION Length(P : T_Pile) RETURN Integer ;
  FUNCTION First(P : T_Pile) RETURN Integer ;
PRIVATE
  TYPE T_Cellule;
  TYPE T_Pile IS ACCESS ALL T_Cellule;
  TYPE T_Cellule IS
    RECORD
      Valeur : Integer;
      Index : Integer;
      Suivant : T_Pile;
    END RECORD;
  END T_Cellule;
END P_Pile;
```

Exercice 2

Énoncé

Créer une classe T_Perso contenant le nom d'un personnage de RPG(jeu de rôle), ses points de vie et sa force. La classe sera accompagnée d'une méthode pour saisir un objet de type T_Perso et d'une seconde pour afficher ses caractéristiques.

Solution

Secret (cliquez pour afficher)

Code : Ada - P_Perso.ads

```
package P_Perso is
  type T_Perso is private;          --éventuellement limited private
  procedure get(P : out T_Perso);
  procedure put(P : in T_Perso);
private
  type T_Perso is record
    Nom : string(1..20);
    PV : Natural := 0;
    Force : Natural := 0;
  end record;
end P_Perso;
```

Code : Ada - P_Perso.adb

```
with ada.text_io;      use ada.text_io;
with ada.integer_text_io; use ada.integer_text_io;

package body P_Perso is
  procedure get(P : out T_Perso) is
  begin
    put("Entrez un nom (moins de 20 lettres) : ");
    get(P.Nom);
    skip_line;
    put("Entrez ses points de vie : ");
    get(P.PV);
    skip_line;
    put("Entrez sa force : ");
    get(P.Force);
    skip_line;
  end get;

  procedure put(P : in T_Perso) is
  begin
    put_line(P.nom);
    put_line(" VIE : " & integer'image(P.PV));
    put_line(" FORCE : " & integer'image(P.Force));
  end put;
end P_Perso;
```

Voilà qui est fait pour notre entrée en matière avec la POO. Rassurez-vous, ce chapitre n'était qu'une entrée en matière. Si l'intérêt de la POO vous semble encore obscur ou que certains points mériteraient des éclaircissement, je vous conseille de continuer la lecture des prochains chapitres durant lesquels nous allons préciser la notion d'objet et de classe. La prochaine étape est la **généricité**, notion déjà présente en Ada83 et dont je vous ai déjà vaguement parlé au cours des précédents chapitres.

En résumé :

- En Programmation orientée objet, les types d'objets sont appelés classes. Chaque classe dispose de son propre package.
- Le terme méthode regroupe les fonctions et les procédures associées à une classe. Toutes les méthodes nécessaires à sa manipulation doivent être fournies avec le package.
- Lorsque l'on programme « orienté objet », il est important d'encapsuler la classe voire certaines de ses méthodes. Pour cela, la classe doit être déclarée comme privée (**PRIVATE**, ou même limitée et privée (**LIMITED PRIVATE**)).
- Le corps d'un package a visibilité sur l'ensemble de ses spécifications, publiques ou privées.
- Un programme ou un second package ne peut avoir accès qu'aux spécifications publiques. La partie publique agit ainsi comme un gardien bloquant l'accès à la partie privée et au corps et transmettant elle-même les informations demandées.

La programmation modulaire III : Généricité

Cela fait déjà plusieurs chapitres que ce terme revient : **générique**. Les packages Ada.Containers.Doubly_Linked_Lists ou Ada.Containers.Vectors étaient génériques. Mais bien avant cela, le package Ada.Numerics.Discrete_Random qui nous permettait de générer des nombres aléatoirement était générique ! Même pour libérer un pointeur, nous avions utilisé une procédure générique : Ada.Unchecked_Deallocation(). Nous cotoyons cette notion depuis la partie II déjà, sans avoir pris beaucoup de temps pour l'expliquer : c'est que cette notion est très présente en Ada. Et pour cause, la norme Ada83 permettait déjà la généricité. Il est donc temps de lever le voile avant de revenir à la programmation orientée objet et à l'héritage.

Généricité : les grandes lignes

Que veut-on faire ?

Avant de parler de packages génériques, il serait bon d'évoquer le problème qui a mené à cette notion de généricité. Vous avez du remarquer au cours de ce tutoriel que vous étiez souvent amenés à récrire les mêmes bouts de code, les mêmes petites fonctions ou procédures qui servent sans arrêt. Par exemple, des procédures pour échanger les valeurs de deux variables, pour afficher le contenu d'un tableau, en extraire le plus petit élément ou encore en effectuer la somme... Bref, ce genre de sous-programme revient régulièrement et les packages ne suffisent pas à résoudre ce problème : notez bon vieux package P_Integer_Array ne résolvait certains de ces soucis que pour des tableaux contenant des Integer, mais pas pour des tableaux de float, de character, de types personnalisés... De même, nous avons créé des packages P_Pile et P_File uniquement pour des Integer, et pour disposer d'une pile de Float ; il faudrait jouer du copier-coller : c'est idiot !

D'où la nécessité de créer des programmes génériques, c'est-à-dire pouvant traiter «toute sorte» de types de données. A-t-on vraiment besoin de connaître le type de deux variables pour les intervertir ? Non bien sûr, tant qu'elles ont le même type ! L'idée est donc venue de proposer aux programmeurs la possibilité de ne rediger un code qu'une seule fois pour ensuite le réemployer rapidement selon les types de données rencontrées.

Plan de bataille

Il est possible de créer des fonctions génériques, des procédures génériques ou encore des packages génériques : nous parlons d'**unités de programmes génériques**. Ces unités de programmes ne seront génériques que parce qu'elles accepteront des paramètres eux-mêmes génériques. Ces paramètres sont en général des types, mais il est également possible d'avoir comme paramètre de généricité une variable ou une autre unité de programme ! Ce dernier cas (plus complexe) sera vu à la fin de ce chapitre. Ces paramètres seront appellés soit **paramètres génériques** soit **paramètres formels**. Pour l'instant, nous allons retenir que pour réaliser une unité de programme générique il faut avant-tout déclarer le (ou les) type(s) génériques qui sera (seront) utilisé(s) par la suite. D'où un premier schéma de déclaration.

Code : Français - PLAN

```
DÉCLARATION DU (OU DES) TYPE(S) GÉNÉRIQUE(S)
SÉPÉIFICATION DE L'UNITÉ DE PROGRAMME GÉNÉRIQUE
CORPS DE L'UNITÉ DE PROGRAMME GÉNÉRIQUE
```



Notez bien ceci : il est **OBLIGATOIRE** d'écrire les spécifications des unités de programme, même pour les fonctions et procédures !!! Et si j'utilise le gras-rouge, ce n'est pas pour rien 😊

Ensuite, une unité de programme générique, seule, ne sert à rien. Il n'est pas possible de l'utiliser directement avec un type concret comme Integer ou Float (on parle de **types effectifs**). Vous devrez préalablement créer une nouvelle unité de programme spécifique au type désiré. Rassurez-vous, cette étape se fera très rapidement : c'est ce que l'on appelle **instanciation**. D'où un plan de bataille modifié :

Code : Français - PLAN

```
DÉCLARATION DU (OU DES) TYPE(S) GÉNÉRIQUE(S)
SÉPÉIFICATION DE L'UNITÉ DE PROGRAMME GÉNÉRIQUE
CORPS DE L'UNITÉ DE PROGRAMME GÉNÉRIQUE
INSTANCIATION D'UNE UNITÉ DE PROGRAMME SPÉCIFIQUE
```

Un dernier point de vocabulaire

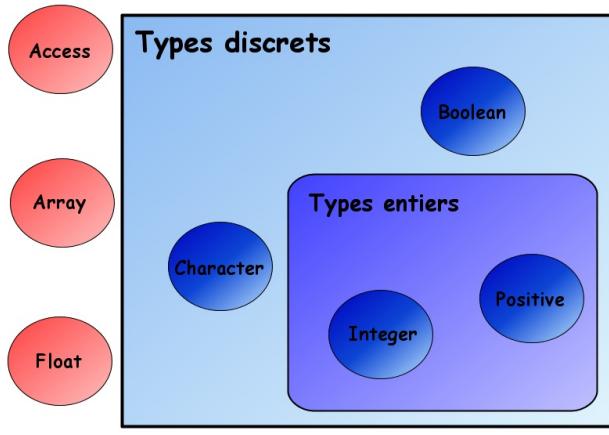


Bon ça y est, on commence ? 😊

Pas encore, nous avons un petit point de théorie à éclaircir. Vous faurez compris, nous allons pouvoir définir des types génériques, seulement les types peuvent être classés en plusieurs catégories qu'il n'est pas toujours évident de distinguer : on dit qu'il existe plusieurs **classes de types** (tiens, encore ce mot «classe» 😊).

Nous avons ainsi les types tableaux, les types flottants, les types entiers, les types pointeurs... mais aussi les types **discrets**. De quoi s'agit-il ? Ce ne sont pas des types qui ne font pas de bruit ! 😊 Non, le terme discret est un terme mathématique signifiant que chaque élément (sauf les extrémités) a un prédécesseur et un successeur. Le type Natural par exemple est un type discret : si vous prenez un élément N au hasard parmi le type Natural (175 par exemple), vous pouvez lui trouver un successeur avec l'instruction Natural'succ(N) (c'est 176) et un prédécesseur avec l'instruction Natural'pred(N) (c'est 174). Et cela marchera pour tous les Natural homos 0 qui n'a pas de prédécesseur et Natural'last qui n'a pas de successeur. Il en va de même pour les tous les types entiers comme Positive ou Integer, pour les types Character, Boolean ou même les types énumérés.

En revanche, le type Float n'est pas un type discret. Prenez un flottant X au hasard (175.0 par exemple). Quel est son successeur ? 176.0 ou 175.1 ? Et pourquoi pas 175.00002 ou 175.0000001 ? De même, les tableaux ou les pointeurs ne sont évidemment pas des types discrets.



Pourquoi vous parlez de cela ? Eh bien parce qu'avant de déclarer des types génériques, il est important de savoir à quelle classe de type il appartiendra : intervertir deux variables ne pose pas de soucis, mais effectuer une addition par 1 ne peut se faire qu'avec un type entier et pas flottant, connaître le successeur d'une variable ne peut se faire qu'avec des types discrets, connaître le n^{ème} élément ne peut se faire qu'avec un type tableau... Bref, faire de la généricité, ce n'est pas faire n'importe quoi : il est important d'indiquer au compilateur quels types seront acceptables pour nos unités de programmes génériques.

Créer et utiliser une méthode générique

Créer une méthode générique

Bon ! Il est temps désormais de voir un cas concret ! Nous allons créer une procédure qui échange deux éléments (nous

l'appellerons Swap, c'est le terme anglais pour Échanger). Voilà à quoi elle ressemblerait :

Code : Ada

```
procedure swap(a,b : in out Positive) is
  c : Positive;
begin
  c:=a;
  a:=b;
  b:=c;
end swap;
```

Mais elle est conçue pour échanger des `Positive` : pas des `Natural` ou des `Integer`, non ! Seulement des `Positive` ! Peut-être pourrions-nous élargir son champ d'action à tous les types entiers, au moins ?

Créer un type générique

Nous allons donc créer un type générique appelé `T_Entier`. Attention, ce type n'existera pas réellement, il ne servira qu'à la réalisation d'une procédure générique (et une seule). Pour cela nous allons devoir ouvrir un bloc `GENERIC` dans la partie réservée aux déclarations :

Code : Ada

```
generic
  type T_Entier is range <>;
```

Notez bien le « `RANGE <>` » ! Le diamant (`<>`) est le symbole indiquant que les informations nécessaires ne seront transmises que plus tard. La combinaison de `RANGE` et du diamant indique plus précisément que le type attendu est un type entier (`Integer`, `Natural`, `Positive`, `Long_Long_Integer`...) et pas flottant ou discret ou que-sais-je encore !

Notez également qu'il n'y a pas d'instruction « `END GENERIC` » ! Pourquoi ? Tout simplement parce que ce type générique ne va servir qu'une seule fois et ce sera pour l'unité de programme que l'on va déclarer ensuite. Ainsi, c'est le terme `FUNCTION`, `PROCEDURE` ou `PACKAGE` qui jouera le rôle du `END GENERIC` et mettra un terme aux déclarations génériques. Le type `T_Entier` doit donc être vu comme un paramètre de l'unité de programme générique qui suivra.

Créer une procédure générique

Comme nous l'avions dit précédemment, la déclaration du type `T_Entier` doit être **immédiatement** suivie de la spécification de la procédure générique, ce qui nous donnera le code suivant :

Code : Ada

```
generic
  type T_Entier is range <>;
procedure Generic_Swap(a,b : in out T_Entier);
```

Comme vous le constaterez, notre procédure utilise désormais le type `T_Entier` déclaré précédemment (d'où son nouveau nom `Generic_Swap`). Cette procédure prend en paramètres deux variables `a` et `b` indiquées entre parenthèses mais aussi un type `T_Entier` indiqué dans sa partie `GENERIC`.

Ne reste plus désormais qu'à rédiger le corps de notre procédure un peu plus loin dans notre programme :

Code : Ada

```
procedure Generic_Swap(a,b : in out T_Entier) is
begin
  c:=a;
  a:=b;
  b:=c;
end Generic_Swap;
```

Utiliser une méthode générique

Instanciation

Mais, j'essaye depuis tout à l'heure ce code, et il ne marche pas :

Code : Ada

```
...
  n : Integer := 3;
  m : Integer := 4;
BEGIN
  Generic_Swap(n,m);
...
```

C'est normal, votre procédure `Generic_Swap()` est faite pour un type générique seulement. Pas pour les `Integer` spécifiquement. Le travail que nous avons fait consistait simplement à rédiger une sorte de « plan de montage », mais nous n'avons pas encore réellement « monté notre meuble ». En programmation, cette étape s'appelle **l'instanciation** : nous allons devoir créer une instance de `Generic_Swap`, c'est-à-dire une nouvelle procédure appelée `Swap_Integer`. Et cela se fait grâce à l'instruction `NEW`:

Code : Ada

```
procedure swap_integer is new Generic_Swap(Integer);
```

Le schéma d'instanciation est relativement simple et sera systématiquement le-même pour toutes les unités de programmes :

procedure				
function	Nom_Unité_Specifique	is new	Nom_Unité_Générique	(Types spécifiques désirés)
package				

Surcharge

Ainsi, le compilateur se chargera de générer lui-même le code de notre procédure `swap_integer`, vous laissant d'avantage de temps pour vous concentrer sur votre programme. Et puis, si vous avez besoin de disposer de procédures `swap_long_long_integer` ou `swap_positive_integer`, il vous suffirait simplement d'écrire :

Code : Ada

```
procedure swap_integer is new Generic_Swap(Integer);
procedure swap_long_long_integer is new
  Generic_Swap(Long_Long_Integer);
procedure swap_positive is new Generic_Swap(Positive);
```

Hop ! Trois procédures générées en seulement trois lignes ! Pas mal non ? Et nous pourrions faire encore mieux, en générant trois instances de `Generic_Swap` portant toutes trois le même nom : `swap` ! Ainsi, en surchargeant la procédure, nous économisons de nombreux caractères à taper ainsi que la nécessité de systématiquement réfléchir aux types employés :

Code : Ada

```
procedure swap is new Generic_Swap(Integer);
procedure swap is new Generic_Swap(Long_Long_Integer);
procedure swap is new Generic_Swap(Positive);
```



Votre procédure générique ne doit pas porter le même nom que ses instances, car le compilateur ne saurait pas faire la

distinction (d'où l'ajout du préfixe "Generic_").

Et désormais, nous pourrons effectuer des inversions à notre guise :

Code : Ada

```
...
generic
    type T_Entier is range <> ;
    procedure Generic_Swap(a,b : in out T_Entier) ;
procedure Generic_Swap(a,b : in out T_Entier) is
    c : T_Entier;
begin
    c:=a;
    a:=b;
    b:=c;
end swap;
procedure swap is new Generic_Swap(Integer);
n : integer := 3;
m : integer := 4;
BEGIN
    swap(n,m);
...
```

Paramètres génériques de types simples et privés

Types génériques simples

Bien, vous connaissez désormais les rudiments de la générnicité en Ada. Toutefois, notre déclaration du type T_Entier implique que vous ne pourrez pas utiliser cette procédure pour des `character` !



Et si je veux qu'elle fonctionne pour les caractères aussi, comment je fais ?

Eh bien il va falloir modifier notre classe de type générique :

- Pour couvrir tous les types **discrets** (entiers mais aussi `character`, `boolean` ou énumérés), on utilisera le diamant seul :

```
type T_Discret is (<>) ;
```

- Pour couvrir les types **réels à virgule flottante** (comme `Float`, `Long_Float`, `Long_Long_Float` ou des types flottants personnalisés), on utilisera la combinaison de `DIGITS` et du diamant :

```
type T_Flottant is digits <> ;
```

- Pour couvrir les types **réels à virgule fixe** (comme `Duration`, `Time` ou les types réels à virgule fixe personnalisés), on utilisera la combinaison de `DELTA` et du diamant, voir de la `DELTA`, `DIGITS` et du diamant (revoir le chapitre Variables III : Gestion des données si besoin) :

```
type T_Fixe is delta <> ;
type T_Fixe is delta <> digits <> ;
```

- Enfin, pour couvrir les types **modulaires** (revoir le chapitre Créez vos propres objets si besoin), on utilisera la combinaison de `MOD` et du diamant :

```
type T_Modulaire is mod <> ;
```

Types génériques privés



Mais, on ne peut pas faire de procédure encore plus générique ? Comme une procédure qui manipulerait des types discrets mais aussi flottants ?

Si, c'est possible. Pour que votre type générique soit «le plus générique possible», il vous reste deux possibilités :

- Si vous souhaitez que votre type bénéficie au moins de l'affectation et de la comparaison, il suffira de le déclarer ainsi :

```
type T_Generique is private ;
```

- Et si vous souhaitez un type le plus générique possible, écrivez :

```
type T_Super_Generique is limited private ;
```



Si votre type générique est déclaré comme `LIMITED PRIVATE`, votre procédure générique ne devra employer ni la comparaison ni l'affectation, ce qui est gênant dans le cas d'un échange de variables.

Paramètres génériques de types composites et programmes

Tableaux génériques

Un premier cas

Pour déclarer un type tableau, nous avons toujours besoin de spécifier les indices : le tableau est-il indexé de 0 à 10, de 3 à 456, de 'a' à 'z', de JANVIER à FÉVRIER... ? Par conséquent, pour déclarer un tableau générique, il faut au minimum deux types généréniques : le type (nécessairement discret) des indices et le type du tableau.

Exemple :

Code : Ada

```
generic
    type T_Indice is (<>) ;
    type T_Tableau is array(T_Indice) of Float ;
    procedure Generic_Swap(T : in out T_Tableau ; i,j : T_Indice) is
begin
    tmp := T(i) ;
    T(i) := T(j) ;
    T(j) := tmp ;
end Generic_Swap ;
```

Notre procédure Generic_Swap dispose alors de deux paramètres formels (`T_Indice` et `T_Tableau`) qu'il faudra spécifier à l'instanciation dans l'ordre de leur déclaration :

Code : Ada

```
...
subtype MesIndices is integer range 1..10 ;
type MesTableaux is array(MesIndices) of Float ;
procedure swap is new Generic_Swap(MesIndices,MesTableaux) ;
T : MesTableaux := (9.0, 8.1, 7.2, 6.3, 5.4, 4.5, 3.6, 2.7, 1.8,
0.9) ;
begin
    swap(T,3,5) ;
...
```

Avec un tableau non contraint

Une première amélioration peut être apportée à ce code. Déclaré ainsi, notre type de tableaux est nécessairement contraint : tous

les tableaux de type MesTableaux sont obligatoirement indexés de 1 à 10. Pour lever cette restriction, il faut tout d'abord modifier notre type formel :

Code : Ada

```
generic
  type T_Indice is (<>) ;
  type T_Tableau is array(T_Indice range <>) of Float ;
  --l'ajout de "range <>" nous laisse la liberté de contraindre
  nos tableaux plus tard
procedure Generic_Swap(T : in out T_Tableau ; i,j : T_Indice) ;
```

Notez bien l'ajout de « **RANGE** <> » après **T_Indice** à la ligne 3 ! On laisse une liberté sur l'intervalle d'indexage des tableaux. Puis, nous pourrons modifier les types effectifs (ceux réellement utilisés par votre programme) :

Code : Ada

```
...
type MesTableaux is array(Integer range <>) of Float ;
--on réécrit "range <>" pour bénéficier d'un type non
constraint !
procedure swap is new Generic_Swap(Integer,MesTableaux) ;
T : MesTableaux(1..6) := (7.2, 6.3, 5.4, 4.5, 3.6, 2.7) ;
--Cette fois, on constraint notre tableau en l'indexant de 1 à
6 ! Étape obligatoire !
begin
  swap(T,3,5) ;
  ...

```

Un tableau entièrement générique

Seconde amélioration : au lieu de disposer d'un type **T_Tableau** contenant des **Float**, nous pourrions créer un type contenant toute sorte d'élément. Cela impliquera d'avoir un troisième paramètre formel : un type **T_Element** :

Code : Ada

```
generic
  type T_Indice is (<>) ;
  type T_Element is private ;
  --Nous aurons besoin de l'affectation pour la procédure
  Generic_Swap donc T_Element ne peut être limited
  type T_Tableau is array(T_Indice range <>) of T_Element;
procedure Generic_Swap(T : in out T_Tableau ; i,j : T_Indice) ;
```

Je ne vous propose pas le code du corps de **Generic_Swap**, j'espère que vous serez capable de le modifier par vous-même (ce n'est pas bien compliqué). En revanche, l'instantiation devra à son tour être modifiée :

Code : Ada

```
type MesTableaux is array(Integer range <>) of Float ;
--Jusque là pas de grosse différence
procedure swap is new Generic_Swap(Integer,Float,MesTableaux) ;
--Il faut indiquer le type des indices + le type des éléments +
le type du tableau
```

L'inconvénient de cette écriture c'est qu'elle n'est pas claire : le type **T_Tableau** doit-il être écrit en premier, en deuxième ou en troisième ? Les éléments du tableau sont des **Integer** ou des **Float** ? Bref, on s'emmêle les pinceaux et il faut régulièrement regarder les spécifications de notre procédure générique pour s'y retrouver. On préférera donc l'écriture suivante.

Code : Ada

```
procedure swap is new Generic_Swap(T_Indice => Integer,
                                    T_Element => Float,
                                    T_Tableau => MesTableaux) ;
--RAPPEL : L'ordre n'a alors plus d'importance ! !
--CONSEIL : Indentez correctement votre code pour plus de
lisibilité
```

Pointeurs génériques

De la même manière, il est possible de créer des types pointeurs génériques. Et comme pour déclarer un type pointeur, il faut absolument connaître le type pointé, cela impliquera d'avoir deux paramètres formels.

Exemple :

Code : Ada

```
generic
  type T_Element is private ;
  type T_Pointeur is access T_Element ;
  procedure Access_Swap(P,Q : in out T_Pointeur) ;
```

D'où l'instantiation suivante :

Code : Ada

```
type MesPointeursPersos is access character ;
procedure Swap is new Access_Swap(character,MesPointeursPersos) ;
-- OU MIEUX :
procedure swap is new Access_Swap(T_Pointeur => MesPointeursPersos,
                                    T_Element  => Character) ;
```

Paramètre de type programme : le cas d'un paramètre **LIMITED PRIVATE**



Bon, j'ai compris pour les différents types tableaux, pointeurs, etc. Mais quel intérêt de créer un type **LIMITED PRIVATE ET GENERIC** ? On ne pourra rien faire ! Ça ne sert à rien pour notre procédure d'échange ! 😊

En effet, nous sommes face à un problème : les types **LIMITED PRIVATE** ne nous autorisent pas les affectations, or nous aurions bien besoin d'un sous-programme pour effectuer cette affectation. Prenons un exemple : nous allons réaliser un package appelé **P_Point** qui sait, lit, affiche... des points et leurs coordonnées. Voici quelle pourrait être sa spécification :

Code : Ada

```
package P_Point is
  type T_Point is limited private ;
  procedure set_x(P : out T_Point ; x : in float) ;
  procedure set_y(P : out T_Point ; y : in float) ;
  function get_x(P : in T_Point) return float ;
  function get_y(P : in T_Point) return float ;
  procedure put(P : in T_Point) ;
  procedure copy(From : in T_Point ; To : out T_Point) ;

  private
    type T_Point is record
      x,y : Float ;
    end record ;
end P_Point ;
```

Ce type **T_Point** est bien **LIMITED PRIVATE** mais son package nous fournit une procédure pour réaliser une copie d'un point

dans un autre. Reprenons désormais notre procédure générique :

Code : Ada

```
generic
  type T_Element is limited private ;
  with procedure copier_A_vers_B(a : in T_Element ; b : out T_Element)
;
procedure Generic_Swap(a,b : in out T_Element) ;
```

Nous ajoutons un nouveau paramètre formel : une procédure appelée `copier_A_vers_B` et qu'il faudra préciser à l'instanciation. Mais revenons avant cela au code source de la procédure `Generic_Swap`. Si elle ne peut pas utiliser le symbole d'affectation `:=`, elle pourra toutefois utiliser cette nouvelle procédure générique `copier_A_vers_B` :

Code : Ada

```
procedure Generic_Swap(a,b : in out T_Element) is
  c : T_Element ;
begin
  copier_A_vers_B(b,c) ;
  copier_A_vers_B(a,b) ;
  copier_A_vers_B(c,a) ;
end generic_swap ;
```

Désormais, si vous désirez une procédure pour échanger deux points, vous devrez préciser, à l'instanciation, le nom de la procédure qui effectuera cette copie de A vers B :

Code : Ada

```
procedure swap is new generic_swap(T_Point,copy) ;
-- OU BIEN
procedure swap is new generic_swap(T_Element      => T_Point,
                                     Copier_A_vers_B => copy) ;
```



Si à l'instanciation vous ne fournissez pas une procédure effectuant une copie exacte, vous vous exposez à un comportement erratique de votre programme.



Il est possible d'avoir plusieurs procédures ou fonctions comme paramètres formels de généricité. Il est même possible de transmettre des opérateurs comme `"+"`, `"*"`, `"<"`...

Une dernière remarque : si vous ne souhaitez pas être obligé de spécifier le nom de la procédure de copie à l'instanciation, il suffit de modifier la ligne :

Code : Ada

```
with procedure copier_A_vers_B(a : in T_Element ; b : out T_Element)
;
```

En y ajoutant un diamant (mais n'oubliez pas de modifier le code source de `Generic_Swap` en conséquence). L'opération d'affectation standard sera alors utilisée.

Code : Ada

```
with procedure copy(a : in T_Element ; b : out T_Element) is <> ;
```

Packages génériques

Exercice

Vous avez dès maintenant appris l'essentiel de ce qu'il y a à savoir sur la généricité en Ada. Mais pour plus de clarté, nous n'avons utilisé qu'une procédure générique. Or, la plupart du temps, vous ne créerez pas un seul programme générique, mais bien plusieurs, de manière à offrir toute une palette d'outil allant de paire avec l'objet générique que vous proposerez. Prenez l'exemple de nos packages `P_Pile` et `P_File` (encore et toujours eux) : quel intérêt y a-t-il à ne proposer qu'une seule procédure générique alors que vous disposez de plusieurs primitives ? Dans 95% des cas, vous devrez donc créer un package générique.

Pour illustrer cette dernière (et courte) partie, et en guise d'exercice final, vous allez donc modifier le package `P_Pile` pour qu'il soit non seulement limité privé, mais également générique ! Ce n'est pas bien compliqué : retenez le schéma que je vous avais donné en début de chapitre.

Code : Autre

```
DÉCLARATION DU (OU DES) TYPE(S) GÉNÉRIQUE(S)
SÉPÉCIFICATION DE L'UNITÉ DE PROGRAMME GÉNÉRIQUE
CORPS DE L'UNITÉ DE PROGRAMME GÉNÉRIQUE
INSTANCIATION D'UNE UNITÉ DE PROGRAMME SPÉCIFIQUE
```

La différence, c'est que le tout se fait dans des fichiers séparés :

Code : Autre - `P_Pile.ads`

```
DÉCLARATION DU (OU DES) TYPE(S) GÉNÉRIQUE(S)
SÉPÉCIFICATION DU PACKAGE GÉNÉRIQUE
```

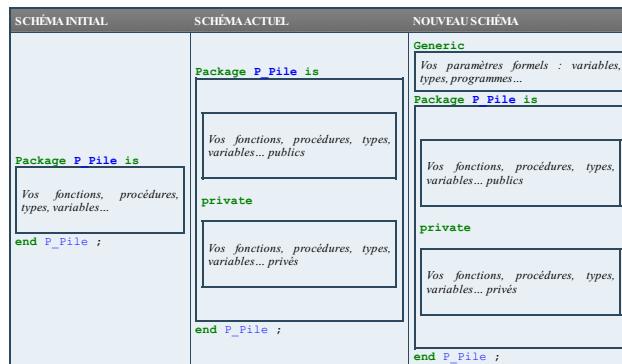
Code : Autre - `P_Pile.adb`

```
CORPS DU PACKAGE GÉNÉRIQUE
```

Code : Autre - `VotreProgramme.adb`

```
INSTANCIATION D'UN PACKAGE SPÉCIFIQUE
```

La structure de votre fichier `ads` va donc devoir encore évoluer :



Vous êtes prêts pour le grand plongeon ? Alors allez-y ! 🎉

Solution

Secret (cliquez pour afficher)

Code : Ada - Fichier P_Pile.ads

```
GENERIC
TYPE T_Element IS PRIVATE ;
PACKAGE P_Pile IS
TYPE T_Pile IS LIMITED PRIVATE ;
PROCEDURE Push (P : IN OUT T_Pile; N : IN T_Element) ;
PROCEDURE Pop (P : IN OUT T_Pile ; N : OUT T_Element) ;
FUNCTION Empty (P : IN T_Pile) RETURN Boolean ;
FUNCTION Length(P : T_Pile) RETURN Integer ;
FUNCTION First(P : T_Pile) RETURN T_Element ;
PRIVATE
TYPE T_Cellule;
TYPE T_Pile IS ACCESS ALL T_Cellule;
TYPE T_Cellule IS
RECORD
Valeur : T_Element ; --On crée une pile générique
Index : Integer;
Suivant : T_Pile;
END RECORD;
END P_Pile;
```

Code : Ada - Fichier P_Pile.adb

```
PACKAGE BODY P_Pile IS
PROCEDURE Push (
P : IN OUT T_Pile;
N : IN T_Element) IS
Cell : T_Cellule;
BEGIN
Cell.Valeur := N ;
IF P /= NULL
THEN
Cell.Index := P.all.Index + 1 ;
Cell.Suivant := P.all'ACCESS ;
ELSE
Cell.Index := 1 ;
END IF ;
P := NEW T_Cellule'(Cell) ;
END Push ;

PROCEDURE Pop (
P : IN OUT T_Pile;
N : OUT T_Element) IS
BEGIN
N := P.all.Valeur ; --ou P.valeur
--P.all est censé exister, ce sera au programmeur final de
le vérifier.
IF P.all.Suivant /= NULL
THEN
P := P.Suivant ;
ELSE
P := NULL ;
END IF ;
END Pop ;

FUNCTION Empty (
P : IN T_Pile)
RETURN Boolean IS
BEGIN
IF P= NULL
THEN
RETURN True ;
ELSE
RETURN False ;
END IF ;
END Empty ;

FUNCTION Length(P : T_Pile) RETURN Integer IS
BEGIN
IF P = NULL
THEN
RETURN 0 ;
ELSE
RETURN P.Index ;
END IF ;
END Length ;

FUNCTION First(P : T_Pile) RETURN T_Element IS
BEGIN
RETURN P.Valeur ;
END First ;
END P_Pile;
```

Application

Vous pouvez désormais créer des piles de `Float`, de tableaux, de `boolean`... Il vous suffira juste d'instancier votre package :

Code : Ada

```
With P_Pile ; --Pas de clause Use, cela n'aurait aucun sens
car P_Pile est générique
procedure MonProgramme is
package P_Pile_Character is new P_Pile(Character) ;
use P_Pile_Character ;
P : T_Pile ;
begin
push(P,'Z') ;
push(P,'#') ;
...
end MonProgramme ;
```



Si vous réalisez deux instantiations du même package `P_Pile`, des confusions sont possibles que le compilateur n'hésitera pas à vous faire remarquer. Regardez l'exemple suivant.

Code : Ada

```
With P_Pile ;
procedure MonProgramme is
package P_Pile_Character is new P_Pile(Character) ;
use P_Pile_Character ;
package P_Pile_Float is new P_Pile(Float) ;
use P_Pile_Float ;
P : T_Pile ; --Est-ce une pile de Float ou une pile de Character ?
begin
...
end MonProgramme ;
```

Privilégiez alors l'écriture pointée afin de préciser l'origine de vos objets et méthodes :

Code : Ada

```
P : P_File_Float.T_Pile ;      --Plus de confusion possible
```

En résumé :

- Comme vous avez pu le constater, la généricité est une notion compliquée mais puissante. Elle vous permettra d'élaborer des outils complexes (comme les piles ou les files) disponibles pour toute une panoplie de type.
- En Ada, une **unité de programme générique** peut être : une fonction, une procédure ou un package. Pour créer une unité de programme générique, il suffit d'écrire le mot-clé **GENERIC** juste avant **FUNCTION**, **PROCEDURE** ou **PACKAGE**. Une unité de programme générique est indiquée avec les termes « **WITH FUNCTION...** », « **WITH PROCEDURE...** » ou « **WITH PACKAGE...** ».
- Il est important de bien classer votre type : ajouter l'est possible qu'avec un type entier, utiliser les attributs '**first**', '**last**', '**succ**' ou '**pred**' n'est possible qu'avec un type discret...
- Pour utiliser une unité de programme générique vous devez d'abord en créer une **instance** avec **NEW** et en fournissant les types que vous souhaitez réellement utiliser.

La programmation modulaire IV : Héritage et dérivation

Après un chapitre sur la généréricité plutôt (voire carrément) théorique, nous enchainons avec un second tout aussi compliqué, mais à combien important. Je dirais même que c'est le chapitre phare de cette partie IV : l'héritage. Et pour cause, l'héritage est, avec l'encapsulation, la propriété fondamentale de la programmation orientée objet.

Pour illustrer ce concept compliqué, nous prendrons, tout au long de ce chapitre, le même exemple : imaginez que nous souhaitions créer un jeu de stratégie médiéval. Votre mission (si vous l'acceptez !) : créer des objets pour modéliser les différentes unités du jeu : chevalier, archer, arbalétrier, catapulte, voleur, écuyer... ainsi que leurs méthodes associées : attaque frontale, défense, tir, bombardement, attaque de flanc...

Pour bien commencer

Héritage : une première approche théorique

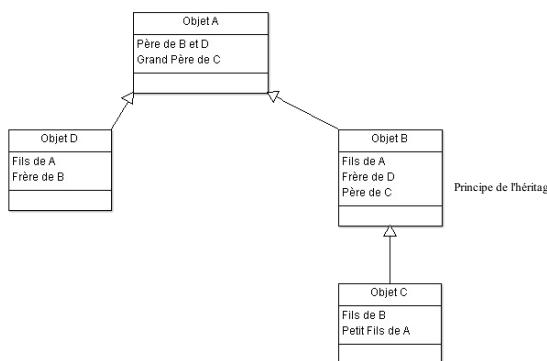
Attention, question métaphysique : qu'est-ce qu'un héritage ?

Eh bien c'est quand papy décède et que les enfants et petits-enfants se partagent le pactole !

Mouais, j'espérais un peu plus de sérieux. Mais vous n'êtes pas si loin de la définition d'héritage en programmation. Non, nos programmes ne vont pas décéder mais ils vont transmettre leurs biens. En effet, dire qu'un objet B hérite d'un objet A, cela revient à dire que B bénéficiera de tous les attributs de A : types, fonctions, procédures, variables... et ce, sans avoir besoin de tout redéfinir.

On dit alors que A est l'objet **père** et B l'objet **fils**. On parle alors d'héritage Père-Fils (ou mère-fille, c'est selon les accords). De même, si un objet C hérite de B, alors il bénéficiera des attributs de B, mais aussi de A par héritage. On dit que C est le fils de B et le petit-fils de A.

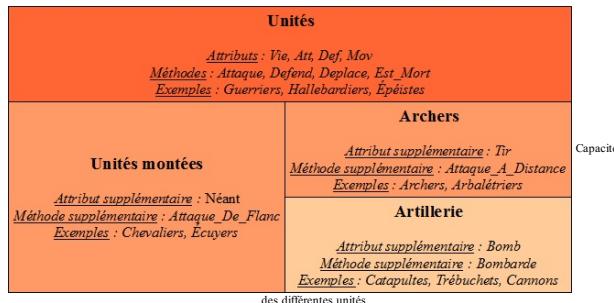
Un dernier exemple : si un deuxième objet D hérite de A, on dira que D et B sont deux objets frères. Ils bénéficient chacun des attributs du père A, mais B ne peut pas bénéficier des attributs de son frère D et réciproquement : il n'y a pas d'héritage entre frères en programmation. Comment cela vous êtes perdus ? Allez, pour ceux qui ont toujours été nuls en généalogie, voici un petit schéma où chaque flèche signifie «hérite de» :



Héritage : une approche par l'exemple

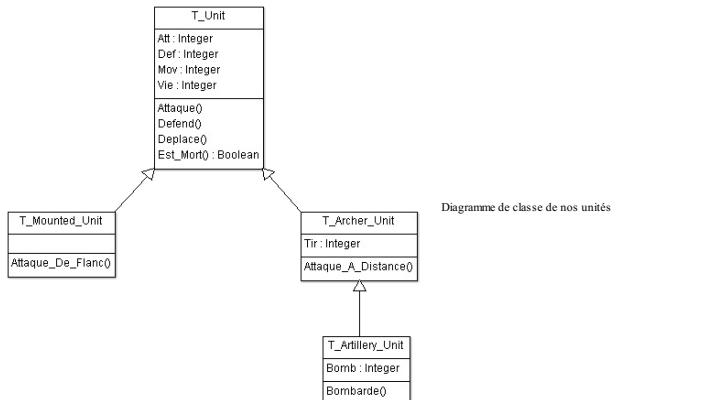
Pour mieux comprendre l'utilité de l'héritage (et avant de découvrir son fonctionnement en Ada), revenons à l'exemple donné en introduction. Nous souhaiterions créer un type d'objet pour représenter des unités militaires pour un jeu de stratégie. Quelles sont les attributs et fonctionnalités de base d'une unité militaire ? Elle doit disposer d'un potentiel d'attaque, de défense et de mouvement. Elle doit également pouvoir attaquer, se défendre et se déplacer.

Toutefois, certaines unités ont des «pouvoirs spéciaux» afin de rendre le jeu plus attrayant : certaines sont montées à cheval pour prendre l'ennemi de vitesse et l'attaquer par le côté ; d'autres ont la capacité d'attaquer à distance à l'aide d'arc, d'arbalètes ou d'engin de siège ; et dans le cas des engins de siège, les unités qui en sont dotées ont alors en plus la possibilité de bombarder l'ennemi pour l'affaiblir avant l'assaut. Résumons par un petit schéma :



Les unités d'artillerie bénéficient des capacités des unités archères, qui elles-mêmes disposent des capacités des Unités classiques (mais l'inverse n'est pas vrai). De même, les unités montées disposent des capacités des unités classiques mais pas de celles des unités archères. On dit alors que tous ces types d'unités font partie de la **classe de type** Unité. Les types Archer et Artillerie font partie de la sous-classe fille Archers, mais pas de la sous-classe sœur Unités montées.

Mais plutôt que ce lourd tableau, nous modéliserons cette situation à l'aide d'un **diagramme de classe UML** comme nous l'avons fait précédemment avec nos objets A, B, C et D :



Chaque case correspond à un type d'objet particulier. La première ligne indique le nom du type ; la seconde ligne fait la liste de ses attributs (notez que les attributs hérités de `T_Unit` ne sont pas réécrits) ; la troisième ligne liste les méthodes supplémentaires. Chacune des flèches signifie toujours « hérite de ... ». Bien ! Maintenant que les idées sont fixées, voyons comment réaliser ce fameux héritage en Ada.

Héritage

Héritage de package simple

Données initiales

Supposons que nous disposions d'un package `P_Unit` contenant le type `T_Unit` et ses méthodes :

```
Code : Ada - P_Unit.ads
package P_Unit is
    type T_Unit is record
        Att, Def, Mov, Vie : Integer := 0;
    end record;

    procedure Attaque(Attaquant, Defenseur : in out T_Unit);
    procedure Defense(Defenseur : in out T_Unit);
    procedure Deplacer(Unite : in out T_Unit);
    function Est_Mort(Unite : in T_Unit) return Boolean;
end P_Unit;
```

Le corps des méthodes sera minimal, pour les besoins du cours :

```
Code : Ada - P_Unit.adb
With ada.text_IO;           Use Ada.Text_IO;
package body P_Unit is
    procedure Attaque(Attaquant, Defenseur : in out T_Unit) is
    begin
        Put_line(" >> Attaque frontale !");
        Defenseur.vie := integer'max(0, Defenseur.vie - Attaquant.att);
    end Attaque;

    procedure Defense(Defenseur : in out T_Unit) is
    begin
        Put_line(" >> Votre unite se defend.");
        Defenseur.def := integer(float(Defenseur.def) * 1.25);
    end Defense;

    procedure Deplacer(Unite : in out T_Unit) is
    begin
        Put_line(" >> Votre unite se deplace");
    end Deplacer;

    function Est_Mort(Unite : in T_Unit) return Boolean is
    begin
        return (Unite.vie <= 0);
    end Est_Mort;
end P_Unit;
```



On n'est pas sensé mettre le type `T_Unit` en `GENERIC` et `PRIVATE` voire `LIMITED PRIVATE` ?

Un type générique ? Non, la générativité n'a rien à voir avec l'héritage. Nous avons besoin d'un type concret ! En revanche, nous devrions encapsuler notre type pour réaliser proprement notre objet mais pour les besoins du cours, nous ne respecterons pas cette règle d'or (pour l'instant seulement). Nous disposons également d'un programme appelé `Strategy` qui crée deux objets `Hallebardier` et `Chevalier` de type `T_Unit` et les fait s'affronter :

```
Code : Ada - Strategy.adb
with P_Unit;           use P_Unit;
with Ada.Text_IO;       use Ada.Text_IO;

procedure Strategy is
    Hallebardier : T_Unit := (Att => 3,
                               Def => 5,
                               Mov => 1,
                               Vie => 10);
    Chevalier     : T_Unit := (Att => 5,
                               Def => 2,
                               Mov => 3,
                               Vie => 8);
begin
    Attaque(Chevalier, Hallebardier);
    if Est_mort(Hallebardier)
        then put_line("Echec et Mat, baby !");
        else put_line("Pas encore !");
    end if;
end Strategy;
```

Un premier package fils

Mais comme `Chevalier` devrait être une unité montée, nous voudrions proposer le choix au joueur entre attaquer et attaquer par le flanc. Nous allons donc créer un package spécifique pour gérer ces unités montées : nous ne nous soucierons pas du type pour l'instant, seulement du package. Nous ne voudrions pas être obligés de recréer l'intégralité du package `P_Unit`, nous allons donc devoir utiliser cette fameuse propriété d'héritage. Attention, ouvrez grand les yeux, ça va aller très vite :

```
Code : Ada - P_Unit-Mounted.ads
package P_Unit.Mounted is
    procedure Attaque_de_flanc(Attaquant, Defenseur : in out T_Unit);
end P_Unit.Mounted;
```

⚠️ TADAAM !!! Cest fini, merci pour le déplacement !

⚠️ ...Attend ! Cest tout ? Mais t'as absolument rien fait ?!

Mais bien sûr que si ! Soyez attentifs au nom : « `PACKAGE P_Unit.Mounted` » ! Le simple fait d'étendre le nom du package père (`P_Unit`) à l'aide d'un point suivi d'un suffixe (`Mounted`) suffit à réaliser un package fils. Simple, n'est-ce pas ? En plus, il est complètement inutile d'écrire « `WITH P_Unit` », puisque notre package hérite de toutes les fonctionnalités de son père (types, méthodes, variables...) !

⚠️ Le nom du package fils doit s'écrire sous la forme `Père . Fils`. En revanche, le nom du fichier correspondant doit s'écrire sous la forme `Père - Fils.adb` (avec un tiret et non un point) pour éviter des confusions avec les extensions de fichier.

Quant au corps de ce package, il sera lui aussi très succinct :

```
Code : Ada - P_Unit-Mounted.adb
With ada.text_IO; Use Ada.Text_IO;

package body P_Unit.Mounted is
    procedure Attaque_de_flanc(Attaquant, Defenseur : in out T_Unit)
    is
        begin
            Put_line(" >> Attaque par le flanc ouest !");
            Defenseur.vie := integer'max(0, Defenseur.vie -
Attaquant.att*Attaquant.mov);
```

```
end Attaque_de_flanc ;
end P_Unit.Mounted ;
```

Utilisation de notre package

Maintenant, revenons à notre programme Strategy. Nous souhaitons laisser le choix au joueur entre Attaque() et Attaque_De_Flanc(), ou plus exactement entre P_Unit.Attaque() et P_Unit.Mounted.Attaque_De_Flanc(). Nous devons donc spécifier ces deux packages, le père et le fils, en en-tête :

Code : Ada

```
with P_Unit ; use P_Unit ;
with P_Unit.Mounted ; use P_Unit.mounted ;
with Ada.Text_IO ;           use Ada.Text_IO ;

procedure Strategy is
    Hallebardier : T_Unit := (Att => 3,
                               Def => 5,
                               Mov => 1,
                               Vie => 10) ;
    Chevalier    : T_Unit := (Att => 5,
                               Def => 2,
                               Mov => 3,
                               Vie => 8) ;
    choix : character ;
begin
    Put("Voulez-vous attaquer frontalement (1) ou par le flanc (2) ?")
    get(choix) ; skip_line ;
    if choix = '1'
    then Attaque(Chevalier,Hallebardier) ;
    else Attaque_de_flanc(Chevalier,Hallebardier) ;
    end if ;
    if Est_mort(Hallebardier)
        then put_line("Echec et Mat, baby !") ;
        else put_line("Pas encore !") ;
    end if ;
end Strategy ;
```

Il n'y a là rien d'extraordinaire, vous l'auriez deviné vous-même.

Deux héritages successifs !

Passons maintenant à nos packages pour archers et artillerie. Le premier doit hériter de P_Unit, mais vous savez faire désormais (n'oubliez pas de réaliser le corps du package également).

Code : Ada - P_Unit-Archer.ads

```
package P_Unit.Archer is
    procedure Attaque_a_distance(Attaquant, Defenseur : in out
                                T_Unit) ;
end P_Unit.Archer ;
```

Code : Ada - P_Unit-Archer.adb

```
With ada.text_IO ;           use Ada.Text_IO ;

package body P_Unit.Archer is
    procedure Attaque_a_distance(Attaquant, Defenseur : in out
                                T_Unit) is
    begin
        Put_line(" >> Une pluie de flèches est décochée ! ");
        Defenseur.vie := integer'max(0,Defenseur.vie - Attaquant.tir)
    end Attaque_a_distance ;
end P_Unit.Archer ;
```

Quant au package pour l'artillerie, nous souhaiterions qu'il hérite de P_Unit.Archer, de manière à ce que les catapultes et trébuchets puissent attaquer, attaquer à distance et en plus bombarder ! Il va donc falloir réutiliser la notation pointée :

Code : Ada - P_Unit-Archer_Artillery.ads

```
package P_Unit.Archer.Artillery is
    procedure Bombarde(Attaquant, Defenseur : in out T_Unit) ;
end P_Unit.Archer.Artillery ;
```

Code : Ada - P_Unit-Archer_Artillery.adb

```
package body P_Unit.Archer.Artillery is
    procedure Bombarde(Attaquant, Defenseur : in out T_Unit) is
    begin
        Put_line(" >> Bombardez-moi tout ça !");
        Defenseur.vie := integer'max(0,Defenseur.vie - Attaquant.bomb)
    end Bombarde ;
end P_Unit.Archer.Artillery ;
```

Et le tour est joué ! Seulement, cela va devenir un peu laborieux d'écrire en en-tête de notre programme :

Code : Ada

```
with P_Unit.Archer.Artillery ;      use P_Unit.Archer.Artillery ;
```

Alors pour les «fainéants du clavier», Ada a tout prévu ! Il y a une instruction de renommage : **RENAMES**. Celle-ci va nous permettre de créer un package qui sera exactement le package P_Unit.Archer.Artillery mais avec un nom plus court :

Code : Ada - P_Artillery.ads

```
With P_Unit.Archer.Artillery ;
package P_Artillery renames P_Unit.Archer.Artillery ;
```

Notez qu'il est complètement inutile de réaliser le corps de ce package P_Artillery, ce n'est qu'un **surnommage**.

Héritage avec des packages privés

 Perso, je ne vois pas vraiment l'intérêt de ce fameux héritage ☺ On aurait très bien pu faire trois packages P_Mounted, P_Archer et P_Artillery avec un «**WITH P_Unit**», et le tour était joué ! Quel intérêt de faire des packages Père-fils ?

L'intérêt est double : primo, cela permet de hiérarchiser notre code et notre travail, ce qui n'est pas négligeable lors de gros projets. Secundo, vous oubliez que nous avons négligé l'encapsulation ! Il n'est pas bon que notre type T_Unit soit public. En fait, nous devrions modifier notre package P_Unit pour privatiser T_Unit et lui fournir une méthode pour l'initialiser :

Code : Ada - P_Unit.ads

```
package P_Unit is
    type T_Unit is private ;
    Function Init(Att, Def, Mov, Vie : Integer := 0) return T_Unit ;
```

```

procedure Attaque(Attaquant, Defenseur : in out T_Unit) ;
procedure Defense(Defenseur : in out T_Unit) ;
procedure Deplacer(Unite : in out T_Unit) ;
function Est_Mort(Unite : in T_Unit) return Boolean ;

private

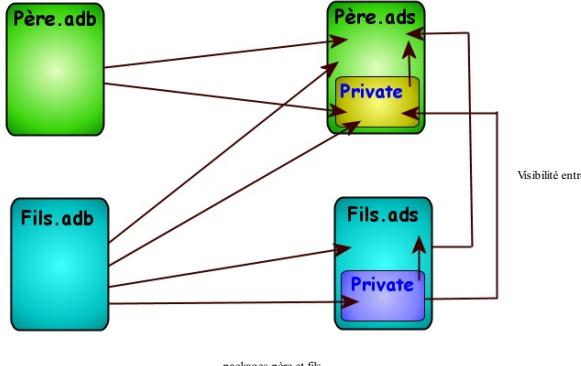
type T_Unit is record
    Att, Def, Mov, Vie : Integer := 0;
end record;

end P_Unit ;

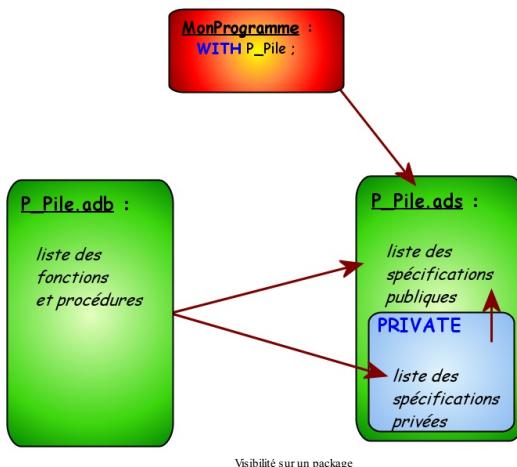
```

 Mais, si `T_Unit` devient privé, alors les procédures `Attaque_De_Flanc()`, `Attaque_a_distance()` ou `Bombarde()` n'y auront plus accès puisqu'elles ne sont pas dans le corps du package `P_Unit` !??!

Tout dépend de la solution que vous adoptez. Si vous faites appel aux instructions `WITH` et pas à l'héritage, ces procédures rencontreront effectivement des problèmes. Mais si vous faites appel à l'héritage alors il n'y aura aucun souci, car les spécifications d'un package fils ont visibilité sur les spécifications publiques et privées du package père ! L'héritage se fait donc également sur la partie `PRIVATE`. L'inverse n'est bien sûr pas vrai : le père n'hérite rien de ses fils. Pour plus de clarté, voici un schéma sur lequel les flèches signifient qu'une partie a la visibilité sur ce qui est écrit dans une autre.



On comprend alors que la propriété d'héritage donne de très nombreux droits d'accès aux packages fils, bien plus que ne le faisaient les clauses de contexte `WITH`. Les packages fils peuvent ainsi manipuler à leur guise les types privés, ce qui ne serait pas possible autrement. Pour rappel, voici le schéma de la visibilité d'un programme sur un package que nous avons déjà vu. La visibilité est nettement restreinte :



Héritage avec des packages génériques

Réalisation

Nous allons considérer pour cette sous-partie, et cette sous-partie seulement, que nos points de vie, d'attaque, de défense... ne sont pas des entiers comme indiqués précédemment mais des types entiers génériques ! Je sais, ça devient compliqué mais il faut que vous vous accrochez ! Nous obtenons donc un nouveau package `P_Unit` :

Code : Ada - `P_Unit.ads`

```

generic
type T_Point is range <> ;
Valeur_Initiale : T_Point ; --Tiens ! Une variable générique !?
package P_Unit is

type T_Unit is private ;

Function Init(Att, Def, Mov, Vie : T_Point := Valeur_Initiale)
return T_Unit ; --Revoilà la variable générique !
procedure Attaque(Attaquant, Defenseur : in out T_Unit) ;
procedure Defense(Defenseur : in out T_Unit) ;
procedure Deplacer(Unite : in out T_Unit) ;
function Est_Mort(Unite : in T_Unit) return Boolean ;

private

type T_Unit is record
    Att, Def, Mov, Vie : T_Point := Valeur_Initiale ; --Encore cette
variable générique !
end record ;

end P_Unit ;

```

Nous avons donc un type d'objet `T_Unit` qui est non seulement privé mais également générique : les points de vie, de défense... sont entiers mais on ne sait rien de plus : sont-ce des `Integer`? Des `Natural`? Des `Positive`?... On n'en sait rien et c'est d'ailleurs pour cela que j'ai du ajouter une variable générique pour connaître la valeur initiale (eh oui, 0 ne fait pas partie des `Positive` par exemple !). Mais la difficulté n'est pas là. Puisque les fils héritent des propriétés du père, si `P_Unit` est générique alors nécessairement tous ses descendants le seront également ! Prendons le cas de `P_Unit.Mounted`, nous allons devoir le modifier ainsi :

Code : Ada

```

generic
package P_Unit.Mounted is
--Mounting mechanism for P_Unit
end P_Unit.Mounted ;

```

```
procedure Attaque_ue_ligne(Attaquant, Defenseur : in out T_Unit);
end P_Unit.Mounted;
```



Tu as oublié de récrire le type `T_Point` et la variable `Valeur_Initiale` après `GENERIC` !

Non, je n'ai rien oublié. Le type `T_Point` et la variable `Valeur_Initiale` ont déjà été défini dans le package père, donc `P_Unit.Mounted` les connaît déjà. On doit simplement réécrire le mot `GENERIC` mais sans aucun paramètre formel (à moins que vous ne voulez en ajouter d'autres bien sûr).

Instanciation

Et c'est à l'instanciation que tout cela devient vraiment (mais alors VRAIMENT) pénible. Si vous souhaitez utiliser `P_Unit.Mounted` vous allez devoir l'instancier, vous vous en doutez ! Sauf qu'il n'a pas de paramètres formels ! Donc il faut préalablement instancier le père :

Code : Ada

```
package MesUnites is new P_Unit(Natural,0);
```

Et pour instancier le fils, il faudra utiliser non plus le package générique père `P_Unit` mais le package que vous venez d'instancier :

Code : Ada

```
package MesUnitesMontees is new MesUnites.Mounted;
```

Je vois que certains commencent à peiner. Alors revenons à des packages non-génériques, cela vaudra mieux.

Dérivation et types étiquetés

Créer un type étiqueté

Nous n'avons fait pour l'instant que réaliser des packages pères-fils avec de nouvelles méthodes mais notre diagramme de classe indiquait que les unités d'archers devaient bénéficier de points de tir et les unités d'artillerie de points de tir et de bombardement.



Diagramme de classe des unités

Comment faire ? Nous avons la possibilité d'utiliser des types structurés polymorphes ou mutants, mais cela va à l'encontre de la logique de nos packages ! Il faudrait que le type `T_Unit` prévoit dès le départ la possibilité de disposer de points de tir ou de bombardement alors que le package `P_Unit` ne les utilise pas. De plus, sur d'importants projets, le programmeur réalisant le package `P_Unit` n'est pas nécessairement le même qui réalise `P_Unit.Archer` ! Les fonctionnalités liées au tir et au bombardement ne le concernent donc pas, ce n'est pas à lui de réfléchir à la façon dont il faut représenter ces capacités. Enfin, les types mutants ou polymorphes impliquent une structure très lourde (souvenez-vous des types `T_Bulletins` et des nombreux `CASE` imbriqués les uns dans les autres) ; difficile de relire un tel code, il sera donc compliqué de le faire évoluer, de le maintenir, voire même simplement de le réaliser.

C'est là qu'interviennent les types étiquetés ou **TAGGED** en Ada. L'idée du type étiqueté est de fournir un type de base `T_Unit` fait pour les unités de base, puis d'en faire des «produits dérivés» appelés `T_Mounted_Unit`, `T_Archer_Unit` ou `T_Artillery_Unit` enrichis par rapport au type initial. Pour cela nous allons d'abord étiqueter notre type `T_Unit`, pour spécifier au compilateur que ce type pourra être dérivé en d'autres types (attention, on ne parle pas ici d'héritage mais de dérivation, nuance) :

Code : Ada

```
package P_Unit is
type T_Unit is tagged private;
function Init(Att, Def, Mov, Vie : Integer := 0) return T_Unit;
procedure Attaque(Attaquant, Defenseur : in out T_Unit);
procedure Defense(Defenseur : in out T_Unit);
procedure Deplacer(Unit : in out T_Unit);
function Est_Mort(Unit : in T_Unit) return Boolean;
private
type T_Unit is tagged record
Att, Def, Mov, Vie : Integer := 0;
end record;
end P_Unit;
```

Chacun aura remarqué l'ajout du mot **TAGGED** pour indiquer l'étiquetage du type. Vous aurez également remarqué que le type `T_Unit` ne fait toujours pas mention des compétences Tir ou Bomb et pour cause. Nous allons maintenant créer un type `T_Archer_Unit` qui dérivera du type `T_Unit` initial :

Code : Ada

```
package P_Unit.Archer is
type T_Archer_Unit is new T_Unit with private;
procedure Attaque_a_distance(Attaquant : in out T_Archer_Unit;
Defenseur : in out T_Unit);
private
type T_Archer_Unit is new T_Unit with record
tir : Integer := 0;
end record;
end P_Unit.Archer;
```

Vous remarquerez que le type `T_Archer_Unit` ne comporte pas de mention **TAGGED**. Normal, c'est un type dérivé d'un type étiqueté, il est donc lui-même étiqueté. L'instruction «`TYPE T_Archer_Unit IS NEW T_Unit ...` » est claire : `T_Archer_Unit` est un nouveau type d'unité, issu de `T_Unit`. L'instruction `WITH RECORD` indique les spécificités de ce nouveau type, les options supplémentaires si vous préférez. À noter que l'instruction `WITH` n'a pas ici le même sens que lorsque nous écrivons `WITH Ada.Text_IO`, c'est là toute la souplesse du langage Ada.



Je vous déconseille d'écrire seulement « `TYPE T_Archer_Unit IS PRIVATE` ». Cela reviendrait à cacher à votre programme Strategy qu'il y a eu dérivation, or nous n'avons pas à le cacher, et vous allez comprendre pourquoi.



Ce code n'est pas encore opérationnel !

Et nos méthodes ?

Méthodes héritées de la classe mère



Mais ??? Comme `T_Archer_Unit` n'est pas un **SUBTYPE** de `T_Unit`, notre procédure `Attaque()` ne va plus marcher avec les archers ! ?!

Eh bien si ! Un type étiqueté hérite automatiquement des méthodes du type père ! C'est comme si nous bénéficiions automatiquement d'une méthode « `PROCEDURE Attaque(Attaquant, Defenseur : IN OUT T_Archer_Unit)` » sans avoir eu besoin de la rédiger. On dit que la méthode est **polymorphe** puisqu'elle accepte plusieurs «formes» de données. Sympa non ?

Méthodes s'appliquant à l'intégralité de la classe



Attend ! Tes en train de me dire que les archers ne pourront attaquer que d'autres archers ? Comment je fais pour qu'un

archer puisse attaquer une unité à pied ou un cavalier ? Je dois faire du copier-coller et surcharger la méthode Attaque ?

En effet, c'est problématique et ça risque d'être long. 😊 Heureusement, vous savez désormais que le type `T_Archer_Unit` est de la classe de type de `T_Unit`, il suffit simplement de revoir notre méthode pour qu'elle s'applique à toute la classe `T_Unit`, c'est-à-dire le type `T_Unit` et sa descendance ! Il suffira que le paramètre `Defenseur` soit accompagné de l'attribut `'class'`:

Code : Ada

```
package P_Unit is
    type T_Unit is tagged private ;
    function Init(Att, Def, Mov, Vie : Integer := 0) return T_Unit ;
    procedure Attaque(Attaquant : in T_Unit ; Defenseur : in out T_Unit'Class) ;
    procedure Defense(Defenseur : in out T_Unit) ;
    procedure Deplacer(Unité : in out T_Unit) ;
    function Est_Mort(Unité : in T_Unit) return Boolean ;
private
    type T_Unit is tagged record
        Att, Def, Mov, Vie : Integer := 0 ;
    end record ;
end P_Unit ;
```

Vous remarquerez que je n'abuse pas de cet attribut puisque tous les types dérivant de `T_Unit` hériteront automatiquement des autres méthodes. De même, nous pouvons modifier notre package `P_Unit.Archer` pour que l'attaque à distance s'applique à toute la classe de type `T_Unit` :

Code : Ada

```
package P_Unit.Archer is
    type T_Archer_Unit is new T_Unit with private ;
    procedure Attaque_a_distance(Attaquant : in T_Archer_Unit ; Defenseur : in out T_Unit'Class) ;
private
    type T_Archer_Unit is new T_Unit with record
        tir : Integer := 0 ;
    end record ;
end P_Unit.Archer ;
```

Grâce à l'attribut `'class'`, nous avons donc indiqué que notre méthode ne s'applique pas qu'à un seul type mais à toute une classe de type. Compris ? Bien, nous allons maintenant créer le type `T_Artillery` en le dérivant de `T_Archer_Unit`. Enfin, quand je dis «*«vous»*», je veux dire «*vous allez créer ce type*» ! Moi, j'ai assez travaillé 😊

Secret (cliquez pour afficher)

Code : Ada

```
package P_Unit.Archer.Artillery is
    type T_Artillery_Unit is new T_Archer_Unit with private ;
    procedure Bombarde(Attaquant : in T_Artillery_Unit ; Defenseur : in out T_Unit'Class) ;
private
    type T_Artillery_Unit is new T_Archer_Unit with record
        bomb : integer := 0 ;
    end record ;
end P_Unit.Archer.Artillery ;
```

Là encore, pas besoin d'étiqueter `T_Artillery_Unit` puisqu'il dérive du type `T_Archer_Unit` qui lui-même dérive du type étiqueté `T_Unit`. De même, notre nouveau type hérite des méthodes des classes `T_Unit` et `T_Archer_Unit`.

Eh ! Mais comment on fait pour `T_Mounted_Unit` vu qu'il n'a aucun attribut de plus que `T_Unit` ?

La réponse en image (ou plutôt en code source):

Code : Ada

```
package P_Unit.Mounted is
    type T_Mounted_Unit is new T_Unit with private ;
    procedure Attaque_de_flanç(Attaquant : in T_Mounted_Unit ; Defenseur : in out T_Unit'Class) ;
private
    type T_Mounted_Unit is new T_Unit with null record ;
end P_Unit.Mounted ;
```

Eh oui, il suffit d'indiquer « `WITH NULL_RECORD` » : c'est-à-dire «*sans aucun enregistrement*». Et n'oubliez pas que notre nouveau type héritera directement des méthodes de la classe mère `T_Unit`, mais pas de celles des classes sœurs.

Surcharge de méthodes

C'est normal que mon code ne compile plus ? GNAT ne veut plus de mon type `T_Archer_Unit` à cause de la fonction `Init` ?? 😊

C'est normal en effet puisque par héritage, la classe de type `T_Archer_Unit` bénéficie d'une méthode « `FUNCTION Init(Att, Def, Mov, Vie : Integer := 0) RETURN T_Archer_Unit` ». Sauf que cette méthode a été prévue à la base pour renvoyer un type structuré contenant 4 attributs : att, def, mov et vie. Or le type `T_Archer_Unit` en contient un cinquième supplémentaire : Tir ! Il y a donc une incohérence et GNAT vous impose donc de remplacer cette méthode (`"Init" must be overridden`). Pour cela, rien de plus simple, il suffit de réécrire cette méthode :

Code : Ada

```
package P_Unit.Archer is
    type T_Archer_Unit is new T_Unit with private ;
    overriding
    function Init(Att, Def, Mov, Vie : Integer := 0) return T_Archer_Unit ;
    procedure Attaque_a_distanç(Attaquant : in T_Archer_Unit ; Defenseur : in out T_Unit'Class) ;
private
    type T_Archer_Unit is new T_Unit with record
        tir : Integer := 0 ;
    end record ;
end P_Unit.Archer ;
```

Le mot-clé `OVERRIDING` apparu avec la norme Ada2005, permet d'indiquer que nous réécrivons la méthode et qu'elle remplace donc la méthode mère. Toutefois, ce mot-clé est optionnel et rien ne serait chamboulé si vous le retriez horsin peut-être la lisibilité de votre code. Notez également qu'aucun attribut `Tir` n'a été ajouté, la méthode devra donc attribuer une valeur par défaut à cet attribut. Pour disposer d'une véritable fonction d'initialisation, il faudra surcharger la méthode :

Code : Ada

```
package P_Unit.Archer is
    type T_Archer_Unit is new T_Unit with private ;
    overriding
    function Init(Att, Def, Mov, Vie : Integer := 0) return T_Archer_Unit ;
    function Init(Att, Def, Mov, Vie, Tir : Integer := 0) return T_Archer_Unit ;
    procedure Attaque_a_distanç(Attaquant : in T_Archer_Unit ; Defenseur : in out T_Unit'Class) ;
private
    type T_Archer_Unit is new T_Unit with record
        tir : Integer := 0 ;
    end record ;
```

```
end P_Unit.Archer ;
```

Notez qu'il n'est pas nécessaire de remplacer cette méthode pour les `T_Mounted_Unit` qui sont identiques aux `T_Unit`, mais que le même travail devra en revanche s'appliquer à la classe de type `T_Artillery_Unit`:

Code : Ada

```
package P_Unit.Archer.Artillery is
    type T_Artillery_Unit is new T_Archer_Unit with private ;
    overriding
    function Init(Att, Def, Mov, Vie, Tir : Integer := 0) return
        T_Artillery_Unit ;
    function Init(Att, Def, Mov, Vie, Tir, Bomb : Integer := 0) return
        T_Artillery_Unit ;
    procedure Bombarde(Attaquant : in T_Artillery_Unit ; Defenseur :
        in out T_Unit'class) ;
    private
        type T_Artillery_Unit is new T_Archer_Unit with record
            bomb : integer := 0 ;
        end record ;
end P_Unit.Archer.Artillery ;
```

Un autre avantage des classes

Vous remarquerez que toutes mes méthodes à deux paramètres ont toujours en premier paramètre l'attaquant. Pourquoi est-ce important ? Eh bien car avec une classe il est possible d'éviter le premier paramètre en écrivant ceci :

Code : Ada

```
Chevalier.attaque(Hallebardier) ;
```

Au lieu du sempiternel et peu parlant :

Code : Ada

```
Attaque(Chevalier,Hallebardier) ;
```

Plus clair non ? Les deux écritures demeurent possibles, mais vous constaterez que l'écriture pointée aura de plus en plus ma préférence (notamment car elle est utilisée dans la plupart des langages orientés objets actuels comme Java...). Pour que cette écriture puisse être possible il faut que votre type soit étiqueté (qu'il définisse une classe) et qu'il coïncide avec le premier paramètre de vos méthodes ! C'est donc pour cela que l'attaquant était mon premier paramètre : il est plus logique d'écrire `Attaquant.attaque(defenseur)` que `defenseur.attaque(attaquant)` ! 

En résumé :

- Pour créer un **objet** (*exemple* : Chevalier) nous devons préalablement créer une classe (*exemple* : `T_Unit` ou `T_Mounted_Unit`). En Ada, une **classe** correspond à un type structuré et étiqueté (**TAGGED RECORD**) et à toute sa descendance.
- Les méthodes s'appliquant à un type **TAGGED RECORD** seront automatiquement dérivées si vous créez un type fils. Mais si vous souhaitez qu'une méthode s'applique à la classe entière (sans effet de dérivation) alors vous devrez utiliser l'attribut `*CLASS`.
- Une classe doit être **encapsulée** et donc de type **PRIVATE** ou **LIMITED PRIVATE**.
- Pour parfaire l'encapsulation, types et méthodes doivent être enfermés dans un **package** spécifique. Les types dérivés sont eux enfermés dans des packages-fils : c'est l'**héritage**.
- Éventuellement, la classe peut être **générique** permettant à d'autres programmeurs d'employer vos outils avec divers types de données.

La programmation modulaire V : Polymorphisme, abstraction et héritage multiple

Ce chapitre est en lien direct avec le précédent ; nous allons approfondir nos connaissances sur la dérivation de type et surtout améliorer notre façon de concevoir nos objets. Vous allez le voir, notre façon de procéder pour concevoir nos unités militaires est encore rudimentaire et n'exploite pas toutes les capacités du langage Ada en matière de POO. Il y a encore une propriété fondamentale de la POO que nous n'avons pas complètement exploitée : le polymorphisme.

Polymorphisme

Méthodes polymorphes

Nous allons continuer à brioler nos packages. Désormais, toutes nos unités peuvent attaquer de front puisque par dérivation du type étiqueté `T_Unit`, toutes bénéficient de la même méthode `attaquer()`. Malheureusement, s'il semble logique qu'un fantassin ou un cavalier puisse attaquer de front, il est plutôt suicidaire d'envoyer un archer ou une catapulte en première ligne. Il faudrait que les types issus de la classe `T_Archer_Unit` disposent d'une méthode `attaquer()` spécifique.

En fait il faudrait surcharger la méthode `attaquer()`, c'est ça ?

Pas exactement. Surcharger la méthode consisterait à écrire une seconde méthode `attaquer()` (distincte) avec des paramètres différents (de par leur nombre, leur type ou leur mode de passage). Or, nous bénéficions déjà, par dérivation de type, d'une méthode «`Attaque(attaquant : IN T_Archer_Unit ; défenseur : OUT T_Unit'class)`». Nous ne souhaitons pas la surcharger mais bien la réécrire.

Bref, tu veux qu'on utilise `OVERRIDING`, c'est cela ?

C'est cela en effet. Notre package `P_Unit.Archer` deviendrait ainsi :

```
Code : Ada - P_Unit.Archer.adb

package P_Unit.Archer is
    type T_Archer_Unit is new T_Unit with private ;
    overriding
        function Init(Att, Def, Mov, Vie : Integer := 0) return
            T_Archer_Unit ;
        function Init(Att, Def, Mov, Vie, Tir : Integer := 0) return
            T_Archer_Unit ;
        procedure Attaque_a_distance(Attaquant : in T_Archer_Unit ;
            Defenseur : in out T_Unit'class) ;
        overriding
        procedure Attaque(Attaquant : in T_Archer_Unit ; Defenseur : in out
            T_Unit'class) ;
    private
        type T_Archer_Unit is new T_Unit with record
            tir : Integer := 0 ;
        end record;
end P_Unit.Archer ;
```

```
Code : Ada - P_Unit.Archer.adb

With ada.text_IO ;           Use Ada.Text_IO ;
package body P_Unit.Archer is
    function Init(Att, Def, Mov, Vie : Integer := 0) return
        T_Archer_Unit is
    begin
        return (Att, Def, Mov, Vie, 0) ;
    end Init ;
    function Init(Att, Def, Mov, Vie, Tir : Integer := 0) return
        T_Archer_Unit is
    begin
        return (Att, Def, Mov, Vie, Tir) ;
    end Init ;
    procedure Attaque_a_distance(Attaquant, Defenseur : in out
        T_Unit) is
    begin
        Put_line(" >> Une pluie de flèches est décochée ! ");
        Defenseur.vie := integer'max(0,Defenseur.vie - Attaquant.tir)
    end Attaque_a_distance ;
    procedure Attaque((Attaquant : in T_Archer_Unit ; Defenseur : in out
        T_Unit'class) is
    begin
        Put_line(" >> Une attaque frontale avec ... des archers ?! Erasez-
        les ! ");
        Defenseur.vie := integer'max(0,Defenseur.vie - 1) ;
    end Attaque ;
end P_Unit.Archer ;
```

En procédant ainsi, notre méthode de classe `Attaque()` aura diverses formes selon que ses paramètres seront de la sous-classe `T_Archer_Unit` ou non. On dit alors que la méthode est **polymorphe**. Quel intérêt ? Créons par exemple un sous-programme tour-joueur :

```
Code : Ada

procedure Tour_Joueur(Joueur1, Joueur2 : in out T_Unit'class) is
    choix : character ;
begin
    put_line("Voulez-vous Attaquer ou Défendre ?");
    get(choix) ; skip_line ;
    if choix='a' or choix='A' then
        Joueur1.attaque(joueur2) ;
    else
        Joueur1.defense(joueur2) ;
    end if ;
end Tour_Joueur ;
```

Que va afficher la console ? >> Attaque frontale ! ou >> Une attaque frontale avec ... des archers ?! Erasez-les ! Eh bien vous n'en savez rien, même au moment de la compilation ! Tout dépendra des choix effectués par l'utilisateur ; selon que Joueur1 sera un archer (ou une artilerie) ou bien un fantassin (ou un cavalier), le programme n'effectuera pas le même travail. Le choix de la méthode `attaquer()` est ainsi reporté au moment de l'utilisation du logiciel, et non au moment de la compilation.

Objets polymorphes

Objets polymorphes non mutants

Poussons le polymorphisme plus loin encore. Nous souhaiterions maintenant proposer un système de héros au joueur. Pas besoin de créer une classe `T_Heros`, le principe est simple : un héros est une unité presque comme les autres, sauf qu'elle est unique et qu'il sera possible au joueur de choisir sa classe : fantassin, archer, cavalier ou... artilleire !!! Nous voudrions donc créer un objet `Massacror` de la classe de type `T_Unit` mais qui aurait la possibilité de devenir de type `T_Mounted_Unit` ou `T_Archer_Unit` selon la volonté du joueur, et ainsi de bénéficier de telle ou telle capacité.

On n'est pas censé ne pas pouvoir changer de type ? T'amères pas de répéter qu'Ada est un langage fortement typé !

En effet, nos objets ne peuvent changer de type ou de classe. Par exemple, GNAT ne tolérera pas que vous écriviez ceci :

```
Code : Ada

...
Massacror : T_Unit ;           --je n'initialise pas
mes objets pour plus de clarté ^^
arbalétrier : T_Archer_Unit ;
chevalier : T_Mounted_Unit ;
choix : character ;
```



```
begin
    put("Que voulez-vous devenir ? Arbalétrier ou Chevalier ?")
    ; get(choix) ; skip_line ;
    if choix = 'a' or choix = 'A'
    then heros := arbalétrier ;
    else heros := chevalier ;
    end if ;
```

Ce qui est bien embêtant. Heureusement, la rigueur du langage Ada n'exclut pas un peu de souplesse ! Et si nous faisons appel à l'attribut `'class'` ? Déclarons Massacror :



Code : Ada

```
Massacror : T_Unit'class ;
```

Problème, GNAT me demande alors d'initialiser mon objet ! Plusieurs choix sont possibles. Vous pouvez utiliser l'une des méthodes d'initialisation, par exemple :

Code : Ada

```
Massacror : T_Unit'class := P_Unit.Archer.Init(150,200,20,200,300)
; --Méthode de "bourrin" !
```

Votre objet sera alors automatiquement défini comme de type `T_Archer_Unit`. Pas terrible pour ce qui est du choix laissé au joueur. Une autre façon est de l'initialiser avec un autre objet :

Code : Ada

```
Massacror : T_Unit'class := Hallebardier ;
```

Mais on retrouve le même problème : Massacror sera du même type que Hallebardier. Vous pouvez certes utiliser des blocs `DECLARE`, mais vous risquez de créer un code lourd, redondant et peu lisible. La solution est davantage de demander à l'utilisateur de faire son choix au moment de la déclaration de votre objet à l'aide d'un nouveau sous-programme `Choix_Unit`. Ce sous-programme renverra non pas un objet de type `T_Unit` mais de la classe `T_Unit'class`:

Code : Ada

```
function Choix_Unit return T_Unit'class is
    choix : character ;
begin
    put_line("Archer, Hallebardier ou Chevalier ?") ;
    get(choix) ; skip_line ;
    if choix='a' or choix='A'
    then return P_Unit.Archer.init(6,1,1,8,7) ;
    elsif choix='H' or choix='h'
    then return P_Unit.init(5,8,1,10) ;
    else return P_Unit.Mounted.init(7,3,2,15) ;
    end if ;
end choix_unit ;
...

Massacror : T_Unit'class := choix_unit ;
```

Ainsi, il est impossible au programmeur de connaître à l'avance le type qui sera renvoyé et donc le type de Massacror ! Génial non ? C'est l'avantage du polymorphisme.



Euh... J'ai comme un doute sur le fait qu'un chevalier puisse lancer des flèches... Comment je fais ensuite pour savoir si je peux utiliser `Attaque_a_distance` ou pas ?

Pas d'affolement, vous pouvez toujours tester l'appartenance de Massacror à tel ou tel type ou classe de type et ainsi éviter les erreurs d'utilisation de méthodes, simplement en écrivant :

Code : Ada

```
if Massacror in T_Archer_Unit'class --Est-ce un archer ou une
artillerie ?
then ...
if Massacror in T_Mounted_Unit --Est-ce une unité montée ?
then ...
```

Objets polymorphes et mutants

Allons toujours plus loin dans le polymorphisme. Pourquoi nos héros ne pourraient-ils pas changer de type ? Un héros, c'est fait pour évoluer au cours du jeu, sinon ça n'a aucun intérêt. Que seraient devenus Songoku, Wolverine, Tintin ou le roi Arthur s'ils n'avaient jamais évolué ? Pourquoi ne pas écrire :



Code : Ada

```
...
Massacror : T_Unit'class := choix_unit ;
begin
    Massacror := choix_unit ;
    Massacror := choix_unit ;
...
```

Seulement là, ce n'est pas le compilateur qui va vous arrêter (rien ne cloche du point de vue du langage), mais votre programme. Tout va bien si vous ne changez pas le type de Massacror. Mais si vous décidez qu'il sera archer, puis chevalier... PATATRA ! `raised CONSTRAINT_ERROR : ##.adic## tag check failed !` Autrement dit, problème lié à l'étiquetage. En effet, après la déclaration-initialisation de votre objet Massacror, celui-ci dispose d'un type bien précis. Le programmeur ne le connaît pas, certes, mais ce type est bien défini, et il n'a pas de raison de changer ! En fait, il n'a surtout pas le droit de changer.



Bref, c'est complètement bloqué. Même l'attribut `'class'` ne sert à rien. On est vite ratrrapé par la rigueur du langage.



Rassurez-vous, il y a un moyen de s'en sortir. L'astuce consiste à faire appel aux pointeurs en créant des pointeurs sur la classe entière (le type racine et toute sa descendance).



Agh ! Noooon ! Pas les pointeurs !

Rassurez-vous, nous n'aurons pas besoin de notions très poussées (pour l'instant) et je vous ai fait faire bien pire. Comme nous l'avons dit, Ada est un langage à fort typage : il doit connaître à la compilation la quantité de mémoire statique qui sera nécessaire au programme. Or, entre un objet de type `T_Unit` et un objet de type `T_Archer_Unit`, il y a une différence notable (la compétence Tir notamment). Impossible de redimensionner durant l'exécution l'emplacement mémoire de notre objet Massacror. À moins de faire appel aux pointeurs et à la mémoire dynamique ! Un pointeur n'étant en définitive qu'une adresse, sa taille ne variera pas, qu'il pointe sur un `T_Unit` ou un `T_Archer_Unit` ! Nous allons pour cela faire de nouveau appel à l'attribut `'class'` :

Code : Ada

```
type T_Heros is access all T_Unit'class ;
Massacror : T_Heros ;
```

Et, ô merveille !, plus aucune initialisation n'est nécessaire ! Toutefois, n'oubliez pas que vous manipulez un pointeur et qu'il peut donc valoir `NULL` ! Redéfinissons donc notre sous-programme `Choix_Unit`

Code : Ada

```

function choix_unit return T_Heros is
    choix : character ;
begin
    put_line("Archer, Hallebardier ou Chevalier ?");
    get(choix); skip_line;
    if choix='A' or choix='a'
        then return new T_Archer_Unit'(init(6,1,1,8,7));
    elsif choix='B' or choix='b'
        then return new T_Unit'(init(5,8,1,10));
        else return new T_Mounted_Unit'(init(7,3,2,15));
    end if;
end choix_unit;

```

Désormais, plus rien ne peut vous arrêter. Vous pouvez changer de type à volonté et surtout, user et abuser du polymorphisme :

Code : Ada

```

Massacror, Vengeator : T_Heros ;
begin
loop
    Massacror := Choix_Unit;
    Vengeator := Choix_Unit;
    Tour_Joueur(Massacror.all,Vengeator.all);
    Tour_Joueur(Vengeator.all,Massacror.all);
end loop;
...

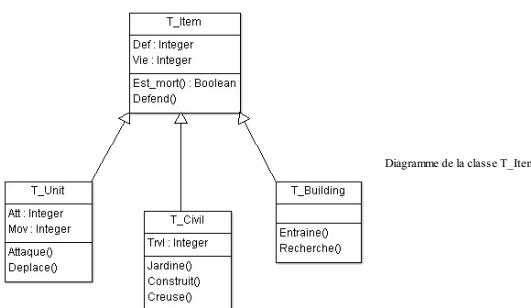
```

Et là, bien malin sera le programmeur qui saura vous dire si le programme affichera >> Une attaque frontale avec des archers ?! Erasez-les ! ou >> Attaque frontale ! Le compilateur lui-même ne saurait vous le dire puisque les objets sont polymorphes et mutants (et donc leur type dépend du bon vouloir du joueur) et que le sous-programme Tour_Joueur fait appel, je vous le rappelle, à la méthode Attaque() qui est elle-même polymorphe !

 Pour plus de lisibilité, je n'ai nullement pris la peine de désallouer la mémoire avant d'en réalloquer aux pointeurs Massacror et Vengeator, mais vous savez bien sûr que c'est une bien mauvaise pratique. Pour un projet plus sérieux, pensez à Ada.Unchecked_Deallocation !

Abstraction Types abstraits

Compliquons encore la chose, nous souhaiterions cette fois créer une super classe T_Item dont toutes les autres hériteraient : les unités civiles (notées T_Civil), les bâtiments (T_Building) et même les unités militaires (T_Unit). A quoi correspondrait la classe T_Item dans notre jeu de stratégie ? Pas à grand chose, je vous l'accorde, mais cela permettrait de mieux hiérarchiser notre code et de regrouper sous une même classe mère les unités militaires du jeu, les unités civiles et les bâtiments ! Et qui dit «même classe mère» dit mêmes méthodes. Cela nous permettra d'utiliser les mêmes méthodes Est_Mort() ou Defend() pour toutes ces classes, et d'user du polymorphisme autant que nous le souhaiterons.



Il est évident qu'aucun objet du jeu ne sera de type T_Item ! Tous les objets appartenant à la classe T_Item seront de type T_Unit, T_Civil, T_Mounted_Unit... mais jamais (JAMAIS) de type T_Item. Ce type restera seulement théorique : on parle alors de type **abstrait** et le terme Ada correspondant (apparu avec la norme Ada95) est tout simplement **ABSTRACT** ! Dès lors, vous savez en doutez, nous devrons dériver ce type car il sera impossible de l'instancier. Conséquence, un type abstrait (**ABSTRACT**) est **obligatoirement étiqueté** (**TAGGED**). Prenons un exemple :

Code : Ada

```

type T_Item is abstract tagged record
    Def, Vie : Integer;
end record;

-- OU ENCORE

type T_Item is abstract tagged private;
private
    type T_Item is abstract tagged record
        Def, Vie : Integer;
    end record;

```

Cela nous obligera à revoir la définition de notre type T_Unit :

Code : Ada

```

type T_Unit is new T_Item with record
    Att, Mov : Integer;
end record;

```



Je répète : il est impossible d'instancier un type abstrait, c'est-à-dire de déclarer un objet de type abstrait !

Très souvent, le type abstrait est même vide : il ne contient aucun attribut et n'existe que comme une racine à l'ensemble des classes :

Code : Ada

```

type T_Item is abstract tagged null record

```

Un autre intérêt des classes abstraites est de pouvoir déclarer un seul type de pointeur pointant sur l'ensemble des sous-classes grâce à l'attribut 'class'. Cette possibilité nous permettra de jouer à fond la carte du polymorphisme :

Code : Ada

```

type T_Ptr_Item is access all T_Item'class;

```

Méthodes abstraites

Qui dit type abstrait dit également, méthodes abstraites. Comme pour les types, il s'agit de méthodes théoriques, inutilisables en fétu : elles auront donc seulement des spécifications et pas de corps (celui-ci sera défini plus tard, avec un type concret) :

Code : Ada

```
procedure defense(defenseur : in out T_Item'Class) is abstract;
function Est_Mort(item : in T_Item'Class) return Boolean is abstract;
;
```

Autre façon de déclarer des méthodes abstraites :

Code : Ada

```
procedure defense(defenseur : in out T_Item'Class) is null;
function Est_Mort(item : in T_Item'Class) return Boolean is null;
```

Le terme **NULL** indique que cette fonction ne fait absolument rien, cela revient à écrire :

Code : Ada

```
procedure defense(defenseur : in out T_Item'Class) is
begin
  null;
end defense;
```

Mais qu'est-ce que l'on peut bien faire avec des méthodes abstraites ? 😊

Eh bien, comme pour les types abstraits, on ne peut rien en faire dans l'état. Le moment venu, lorsque vous disposerez d'un type concret ou mieux, d'une classe de types concrète, il vous faudra réaliser le corps de cette méthode dans la récipient à l'aide de l'instruction **OVERRIDING** (c'est pourquoi certains langages parlent plutôt de méthodes retardées). Ces méthodes abstraites nous permettent d'établir une sorte de «plan de montage», une super-méthode pour super-classe, applicable tout autant aux bâtiments qu'aux unités civiles ou militaires. Elles nous évitent de disposer de méthodes distinctes et incompatibles d'un type à l'autre. Bref, ces méthodes abstraites permettent de conserver le paradigme de polymorphisme (et c'est très important en POO). Tout cela nous amène au package suivant (attention, il n'y a pas de corps pour ce package):

Code : Ada - P_Item.ads

```
package P_Item is
  type T_Item is abstract tagged private;
  procedure Defense(I : in out T_Item'Class) is abstract;
  function Est_Mort(I : T_Item'Class) return Boolean is abstract;
private
  type T_Item is abstract tagged record
    def,vie : integer;
  end record;
end P_Item;
```

Et nous oblige à modifier nos anciens packages.

Secret (cliquez pour afficher)

Code : Ada - P_Item-Unit

```
package P_Item.Unit is
  type T_Unit is new T_Item with private;
  procedure Init(Units : out T_Unit'Class; Att, Def, Mov, Vie : Integer := 0);
  procedure Attaque(Attaquant : in T_Unit'Class; Defenseur : in out T_Unit'Class);
  procedure Defense(Defenseur : in out T_Item'Class); --Réalisation
  --Concrète de la méthode Defense
  procedure Déplacer(Unité : in out T_Unit'Class);
  function Est_Mort(Unité : in T_Item'Class) return Boolean; --Réalisation concrète de la méthode Est_Mort
private
  type T_Unit is new T_Item with record
    Att, Mov : Integer := 0;
  end record;
end P_Item.Unit;
```

Code : Ada - P_Item-Unit-Archer.ads

```
package P_Item.Unit.Archer is
  type T_Archer_Unit is new T_Unit with private;
  overriding
  function Init(Att, Def, Mov, Vie : Integer := 0) return T_Archer_Unit;
  function Init(Att, Def, Mov, Vie, Tir : Integer := 0) return T_Archer_Unit;
  procedure Attaque_a_distance(Attaquant, Defenseur : in out T_Unit);
private
  type T_Archer_Unit is new T_Unit with record
    Att, Def, Mov, Vie : Integer := 0;
    end record;
end P_Item.Unit.Archer;
```

Code : Ada - P_Item-Unit-Archer-Artillery.ads

```
package P_Item.Unit.Archer.Artillery is
  type T_Artillery_Unit is new T_Archer_Unit with private;
  overriding
  function Init(Att, Def, Mov, Vie, Tir : Integer := 0) return T_Artillery_Unit;
  function Init(Att, Def, Mov, Vie, Tir, Bomb : Integer := 0) return T_Artillery_Unit;
  procedure Bombarde(Attaquant : in out T_Artillery_Unit'Class; Defenseur : in out T_Unit'Class);
private
  type T_Artillery_Unit is new T_Archer_Unit with record
    Att, Def, Mov, Vie, Tir : Integer := 0;
    Bomb : integer := 0;
  end record;
end P_Item.Unit.Archer.Artillery;
```

Code : Ada - P_Item-Unit-Mounted.ads

```
package P_Item.Unit.Mounted is
  type T_Mounted_Unit is new T_Unit with private;
  procedure Attaque_de_flanç(Attaquant, Defenseur : in out T_Unit'Class);
private
  type T_Mounted_Unit is new T_Unit with null record;
end P_Item.Unit.Mounted;
```

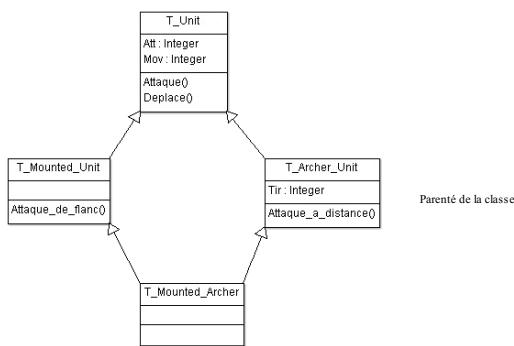


S'il vous souhaitez que les méthodes `defense` et `Est_Mort` soient les mêmes pour les types `T_Unit`, `T_Civil`, `T_Building`... il est toujours possible de déclarer des méthodes non abstraites ayant un paramètre de type `T_Item`. Et bien sûr, votre type `T_Item` ne doit pas être vide. Il n'y a pas d'obligation à créer des méthodes abstraites.

Héritage multiple

Comment Ada gère-t-il l'héritage multiple ?

Nous souhaiterions désormais créer une nouvelle classe d'objet : `T_Mounted_Archer` ! Pour jouer avec des archers montés disposant à la fois des capacités des archers mais également de celles des unités montées (des cavaliers huns ou mongols par exemple 😊). Cette classe dériverait des classes `T_Archer_Unit` et `T_Mounted_Unit` comme l'indique le diagramme de classe ci-dessous :



T_Mounted_Archer

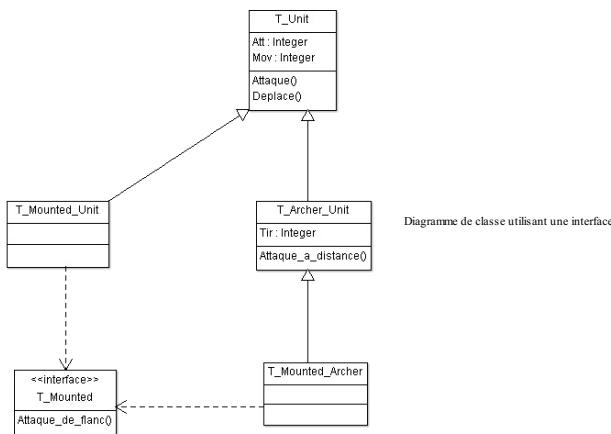
Le seul soucis... c'est qu'en Ada, comme en Java, il n'y a pas d'héritage multiple !

Hein ??? Mais y a tromperie sur la marchandise ! C'était marqué dans le titre : héritage multiple. Et pourquoi tu nous parles du langage Java ? Je comptais pas tout réapprendre maintenant que j'ai lu presque tout ce chapitre !

Du calme, du calme... vous devez comprendre au préalable que le diagramme ci-dessus pose un sérieux problème (appelé problème du diamant en maison de sa forme en losange). Les types T_Mounted_Unit et T_Archer_Unit héritent tous deux de la classe de type T_Unit. Ils héritent donc de la méthode Attaque(). Oui, mais celle-ci a été redéfinie pour les archers, souvenez-vous ! Et puisque T_Mounted_Archer devrait hériter des méthodes de T_Mounted_Unit et de T_Archer_Unit, qui doit afficher la méthode Attaque() pour un archer monté ? >> Une attaque frontale avec... des archers ?! Erasez-les ! ou >> Attaque frontale ! ? Il y a un sérieux conflit d'héritage alors que fatouf major de langage Ada est censé être la fiabilité. Voilà pourquoi l'héritage multiple n'est pas autorisé en Ada comme en Java.

Et si je vous parle du langage Java, c'est parce qu'il s'agit d'un langage entièrement tourné vers la Programmation Orientée Objet : c'est à lui notamment que je pensais quand je vous disais que certains langages comportaient un mot clé **class**. En revanche, le langage Ada, même s'il dispose de très bons atouts pour aborder la POO, n'était pas conçu pour cela à l'origine. La norme Ada95 aura résolu cette lacune grâce à l'introduction du mot clé **TAGGED** et de l'attribut '**class**', entre autres. Mais concernant l'héritage multiple et les risques de sécurité qu'il pouvait entraîner, point de solutions. Et c'est du côté du langage Java qu'est venue l'inspiration. Ce dernier n'admet pas non plus d'héritage multiple pour ses **class**, mais dispose d'une entité supplémentaire : **INTERFACE** ! La norme Ada2005 s'en est inspirée pour fournir à son tour des types **INTERFACE**.

Qu'est-ce qu'une **INTERFACE** ? C'est simplement un type super-abstrait disposant de méthodes **obligatoirement abstraites** et qui peut être hérité plusieurs fois par une classe (et sans risques). Nous allons donc créer un type interface **T_Mounted** qui bénéficiera d'une méthode **Attaque_de_flanc()** et dont hériteront les types **T_Mounted_Unit** et **T_Mounted_Archer**. Ce qui nous donnera le diagramme suivant :



Vous aurez remarqué que les flèches d'héritage sont en pointillés pour les interfaces et que leur nom est précédé du terme «<<interface>>». On ne dit d'ailleurs pas «hériter d'une interface» mais «**implémenter une interface**».

Réaliser des interfaces Ada

Avec une seule interface

Venons-en maintenant au code :

Code : Ada

```

with P_Item.Unit ;      use P_Item.Unit ;

package I_Mounted is
    type T_Mounted is interface ;
        procedure Attaque_de_flanc(Attaquant : in T_Mounted ; Defenseur :
    in out T_Unit'class) is abstract ;
    end I_Mounted ;

```

Et le corps du package ?

Le corps ? Quel corps ? Le type **T_Mounted** étant une **INTERFACE**, il est 100% abstrait et ses méthodes associées (comme **Attaque_de_flanc()**) doivent **obligatoirement** être abstraites. Donc ce package ne peut pas avoir de corps. De plus, il est recommandé de créer un package spécifique pour votre type interface si vous ne voulez pas que GNAT réchigne à compiler. C'est d'ailleurs pour cela que mon package s'appelle **I_Mounted** et non **P_Mounted** (pour **INTERFACE** bien sûr).

Bien ! Notre interface et ses méthodes étant créées, nous pouvons revenir aux spécifications de notre type **T_Mounted_Unit** qui doit désormais implémenter notre interface :

Code : Ada - P_Item-Unit-Mounted.ads

```

With I_Mounted ;      Use I_Mounted ;

package P_Item.Unit.Mounted is
    type T_Mounted_Unit is new T_Unit and T_Mounted with private ;
    overriding
        procedure Attaque_de_flanc(Attaquant : in T_Mounted_Unit ;
    Defenseur : in out T_Unit'class) ;
    private
        type T_Mounted_Unit is new T_Unit and T_Mounted with null record ;
    end P_Item.Unit.Mounted ;

```

Il n'est pas utile de modifier le corps de notre package puisque la procédure `Attaque_de_flanc()` existait déjà. Comme vous avez du vous en rendre compte, il suffit d'ajouter « `AND T_Mounted` » à la définition de notre type pour implémenter l'interface.

 Voici quelques types dérivés ne peuvent pas dériver d'une interface, seulement d'un type étiqueté. L'ordre de déclaration est important : d'abord, le type étiqueté père puis la ou les interfaces. Ce qui nous donne le schéma suivant : « `type MonNouveauType is new TypeEtiquetéPère and InterfaceMère1 and InterfaceMère2 and... with...` »

Venons-en maintenant à notre objectif principal : créer un type `T_Mounted_Archer`.

Code : Ada - P_Item-Unit-Archer-Mounted.ads

```
With I_Mounted ;      Use I_Mounted ;

package P_Item.Unit.Archer.Mounted is
type T_Mounted_Archer is new T_Archer_Unit and T_Mounted with
private :
overriding
procedure Attaque_de_flanc(Attaquant : in T_Mounted_Archer ;
Defenseur : in out T_Unit'class) ;
private
type T_Mounted_Archer is new T_Archer_Unit and T_Mounted with
null record ;
end P_Item.Unit.Archer.Mounted ;
```

Code : Ada - P_Item-Unit-Archer-Mounted.adb

```
With ada.text_IO ;           Use Ada.Text_IO ;

package body P_Item.Unit.Archer.Mounted is
procedure Attaque_de_flanc(Attaquant : in T_Mounted_Archer ;
Defenseur : in out T_Unit'class) is
begin
Put_line(" >> Découchez vos flèches sur le flanc ouest !") ;
Defenseur.vie := integer'max(0,Defenseur.vie -
Attaquant.tir'Attaquant.mov) ;
end Attaque_de_flanc ;
end P_Item.Unit.Archer.Mounted ;
```

Le code est finalement très similaire à celui des unités montées, à ceci près que vous devez écrire le corps du package cette fois. Remarquez que j'ai changé la méthode de calcul ainsi que le texte affiché, mais vous pouvez bien sûr écrire exactement le même code que pour les `T_Mounted_Unit`.

 Mais quel intérêt de réécrire la même chose ? Je pensais que l'héritage nous permettait justement d'éviter cette redondance de code ?

J'avoue avoir eu moi aussi du mal à comprendre cela à mes débuts. Mais le but principal des types abstraits et des interfaces n'est pas de limiter la redondance du code, mais de permettre le polymorphisme. Sans interface, nous aurions deux méthodes `Attaque_de_flanc` distinctes ; avec l'interface, ces deux méthodes n'en forme plus qu'une, mais polymorphe ! Souvenez-vous de nos héros ; ils pouvaient changer de classe sans que le compilateur ne le sache. Il est donc important de privilier une méthode polymorphe plutôt qu'une méthode simplement surchargée.

Avec plusieurs interfaces

Dans le même esprit, pourquoi ne pas créer une interface `I_Archer` ? Ainsi, les archers montés hériteront directement de `T_Unit` et implémenteraient `I_Mounted` et `I_Archer`. La réalisation de l'interface `I_Archer` n'est pas très compliquée :

Code : Ada

```
with P_Item.Unit ;      use P_Item.Unit ;

package I_Archer is
type T_Archer is interface ;
procedure Attaque_a_distance(Attaquant : in T_Archer ; Defenseur : in out T_Unit'class) is abstract ;
end I_Archer ;
```

Que devient alors notre type `T_Archer_Unit` ? Là encore, rien de bien compliqué :

Code : Ada - P_Item-Unit-Archer.ads

```
With I_Archer ;      Use I_Archer ;

package P_Item.Unit.Archer is
type T_Archer_Unit is new T_Unit and T_Archer with private :
overriding
function Init(Att, Def, Mov, Vie : Integer := 0) return T_Archer_Unit ;
function Init(Att, Def, Mov, Vie, Tir : Integer := 0) return T_Archer_Unit ;
-- Réécriture de la méthode héritée de l'interface I_Archer
overriding
procedure Attaque_a_distance(Attaquant : in T_Archer_Unit ;
Defenseur : in out T_Unit'class) ;
-- Réécriture de la méthode héritée de la classe T_Unit
overriding
procedure Attaque(Attaquant : in T_Archer_Unit ; Defenseur : in out T_Unit'class) ;
private
type T_Archer_Unit is new T_Unit and T_Archer with record
Tir : integer ;
end record ;
end P_Item.Unit.Mounted ;
```

Mais ce qui nous intéresse vraiment, c'est notre type `T_Mounted_Archer` :

Code : Ada - P_Item-Unit-Archer-Mounted.ads

```
With I_Mounted ;      Use I_Mounted ;

package P_Item.Unit.Archer.Mounted is
type T_Mounted_Archer is new T_Archer_Unit and T_Mounted and
I_Archer with private :
overriding
procedure Attaque_de_flanc(Attaquant : in T_Mounted_Archer ;
Defenseur : in out T_Unit'class) ;
overriding
procedure Attaque_a_distance(Attaquant : in T_Mounted_Archer ;
Defenseur : in out T_Unit'class) ;
private
type T_Mounted_Archer is new T_Archer_Unit and T_Mounted and
I_Archer with null record ;
end P_Item.Unit.Archer.Mounted ;
```

Code : Ada - P_Item-Unit-Archer-Mounted.adb

```
With ada.text_IO ;           Use Ada.Text_IO ;

package body P_Item.Unit.Archer.Mounted is
procedure Attaque_de_flanc(Attaquant : in T_Mounted_Archer ;
Defenseur : in out T_Unit'class) is
begin
Put_line(" >> Découchez vos flèches sur le flanc ouest !") ;
Defenseur.vie := integer'max(0,Defenseur.vie -
Attaquant.tir'Attaquant.mov) ;
```

```

end Attaque_de_flanc ;

procedure Attaque_a_distance(Attaquant : in T_Mounted_Archer ;
Defenseur : in out T_Unit'class) is
begin
Put_line(" >> Parez ... Tirez !");
Defenseur.vie := integer'max(0,Defenseur.vie - Attaquant.tir);
end Attaque_a_distance ;
end P_Item.Unit.Archer.Mounted ;

```

Comme vous pouvez le constater, mon type `T_Mounted_Archer` hérite désormais ses multiples attributs du type `T_Archer_Unit` et de deux interfaces `I_Archer` et `I_Mounted`. Il est ainsi possible de multiplier les implémentations : pourquoi ne pas imaginer des classes implémentant 4 ou 5 interfaces ?

Interface implémentant une autre interface

Et nos artilleries ? Nous pourrions également créer une interface `I_Artillery` ! Et comme les artilleries héritent des archers, notre interface `I_Artillery` hériterait de `I_Archer`.



Bien sûr ! Tout ce que vous avez à faire, c'est de lui adjondre de nouvelles méthodes (abstraites bien sûr). Cela se fait très facilement. Regardez ce que cela donnerait avec nos artilleries :

Code : Ada

```

with I_Archer ;      use I_Archer ;
with P_Item.Unit ;   use P_Item.Unit ;

package I_Artillery is
  type T_Artillery is interface and T_Archer ;
  procedure Bombarde(Attaquant : in T_Artillery ; Defenseur : in
  out T_Unit'class) ;
end I_Artillery ;

```

Initiale de préciser que `T_Artillery` hérite de la méthode abstraite `Attaque_a_distance()` et qu'elle pourrait hériter de plusieurs interfaces (si l'envisagez-vous prenait). Bien. Avant de clore ce chapitre, il est temps de faire un point sur notre diagramme de classe. Celui-ci a en effet bien changé depuis le début du chapitre précédent. Je vous fais un résumé de la situation ?

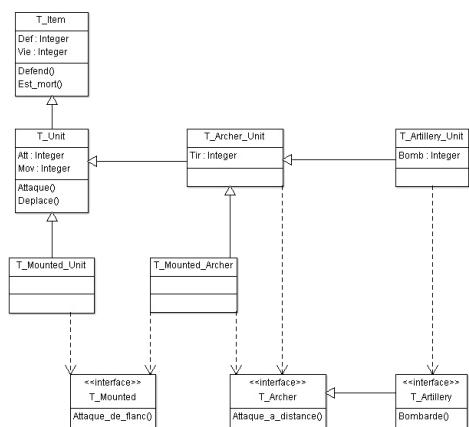


Diagramme de classe final du projet

Fatigués ? C'est normal, vous venez de lire un chapitre intense. Polymorphisme et abstraction ne sont pas des notions évidentes à appréhender. Rassurez-vous, nous en aurons bientôt terminé avec la POO. Il nous reste toutefois un dernier chapitre à aborder avant de nous faire la main sur un nouveau TP : la finalisation et les types contrôlés. Ce chapitre sera bien plus court que les précédents. Alors, encore un peu de courage !

En résumé :

- Pour qu'une même méthode ait des effets différents selon les sous-classes, vous devez créer une méthode **polymorphe**. Il faudra alors réécrire la méthode en précisant : « **OVERRIDING** ».
- Afin de mieux hiérarchiser vos classes, de regrouper sous une même classe différents types tout en conservant les propriétés de polymorphisme, faites appel à un type **abstrait** (**ABSTRACT**). Mais aucun objet ne peut être de type abstrait !
- Pour accompagner votre type abstrait, vous pouvez créer des méthodes abstraites ou non. Les méthodes abstrates devront ensuite être réécrites, les autres seront automatiquement dérivées pour les classes filles.
- En Ada, l'héritage multiple est contourné grâce aux **interfaces**. N'importe quelle classe peut implémenter plusieurs interfaces, ce qui permet de garantir le polymorphisme des méthodes.

La programmation modulaire VI : Finalisation et types contrôlés

Il y a un point dont je vous ai parlé très tôt dans cette partie et sur lequel je ne me suis pas beaucoup étendu (pour ne pas dire pas du tout). Je vous avais dit que procédures et fonctions portaient le nom de méthodes en POO. J'avais également ajouté que certaines méthodes possédaient un nom particulier : constructeur, déestructeur, modifieur... C'est à ces trois méthodes spécifiques que nous allons nous intéresser désormais. Pour cela, nous prolongerons (encore une fois) l'exemple vu au chapitre précédent des héros pouvant être de type `T_Unit`, `T_Mounted_Unit` ou `T_Archer_Unit`. Ce sera également l'occasion de revenir sur un oubli (volontaire rassurez-vous).

Objectifs et prérequis

De quoi parle-t-on exactement ?

Moi je veux bien que tu me parles de constructeur mais ça sert à quoi ? Qu'est-ce qu'elles ont de particuliers tes méthodes ?

Constructeur

Par **constructeur**, on entend une méthode qui va initialiser nos objets. Nos types étant généralement `PRIVATE`, nous avons été obligés de fournir une (ou plusieurs) méthode(s) `Init()` avec nos classes. C'est ce que l'on appelle un constructeur. Mais que se passe-t-il si l'on n'initialise pas un objet de type `T_Unit` par exemple ? Eh bien, le programme va réserver un espace en mémoire pour enregistrer notre objet, mais cet espace ne contiendra aucune information valide. Et si nous sommes amenés à lire cet objet, nous ferons alors lamentablement planter notre ordinateur. Il est donc important d'initialiser nos objets. Une solution consiste à prendre l'habitude de le faire systématiquement et de faire en sorte que les personnes qui travaillent avec vous le fasse elles-aussi systématiquement. Mais ce qui serait préférable, ce serait que nos objets s'initialisent d'eux-mêmes (cela éviterait les oubli^{smiley face}). Nous allons justement rédiger des constructeurs par défaut "automatiques".

Destructeur

Je vous avais expliqué lors du chapitre sur les pointeurs qu'il était important de désallouer la mémoire lorsqu'un pointeur ne servait plus, tout en spécifiant que cette opération était risquée (problème de pointages multiples). Ainsi au chapitre précédent, je vous avais fourni un code similaire à ceci :

Code : Ada

```
...
type T_Heros is access all T_Unit'class;
Massacror, Vengeator : T_Heros;
begin
  loop
    Massacror := Choix.Unite;
    Vengeator := Choix.Unite;
    Tour_Joueur(Massacror.all,Vengeator.all);
    Tour_Joueur(Vengeator.all,Massacror.all);
  end loop;
end;
```

Supposons qu'il s'agisse là non pas du programme principal mais d'un sous-programme. Que se passe-t-il en mémoire après la ligne 11 ? Les pointeurs `Massacror` et `Vengeator` terminent leur portée : ils arrivent au bout du programme dans lequel ils avaient été définis. Les emplacements mémoires liés à ces deux pointeurs sont donc libérés. Oui mais un pointeur n'est jamais qu'une adresse ! C'est bien beau de se débarrasser des adresses mais qu'advient-il des données situées à ces fameuses adresses pointées ? Mystère. Certains compilateurs Ada disposent de ce que l'on appelle un **ramasse-miette** (ou **garbage collector**) qui se charge de libérer les emplacements mémoires dont on a perdu l'adresse, mais ce n'est nullement une obligation imposée par la norme ! Donc mieux vaut ne pas compter dessus. Ce qu'il faut faire, comme vous le savez désormais, c'est faire appel au package `Ada.Unchecked_Deallocation`:

Code : Ada

```
With Ada.Unchecked_Deallocation;
...
type T_Heros is access all T_Unit'class;
procedure free           is new Ada.Unchecked_Deallocation(T_Unit'class,T_Heros);
begin
  Massacror, Vengeator : T_Heros;
  loop
    Massacror := Choix.Unite;
    Vengeator := Choix.Unite;
    Tour_Joueur(Massacror.all,Vengeator.all);
    Tour_Joueur(Vengeator.all,Massacror.all);
  end loop;
  free(Massacror);
  free(Vengeator);
end;
```

Cette procédure est lourde, et il serait plus pratique que cette désallocation se fasse automatiquement. Ce sera donc le rôle des **destructeurs** de mettre fin proprement (et automatiquement) à la vie d'un objet dès lors qu'il aura atteint la fin de sa portée.

Modifieur

Les **modificateurs**, quant à eux, ont pour but de gérer les affectations. Quel intérêt ? Eh bien cela vous permettra de mieux contrôler les opérations d'affectations. Que se passerait-il si un utilisateur entrât 0H 75Min 200s pour un temps, 35/13/2001 pour une date ? Rien de bon, c'est sûr. Que se passerait-il si un programmeur, utilisant vos packages sans en connaître le détail, écrivait « `Massacror := Vengeator` » ou « `pile1 := pile2` » (voir chapitre sur les **TAD**) ? Inconscient qu'il manipule en réalité des pointeurs, il affecterait une adresse pensant affecter des données, prenant ainsi des risques quant à la sécurité de son programme. Bref, il est bon dans certains cas de maîtriser les opérations d'affectation sans pour autant les interdire (contrairement aux types `LIMITED PRIVATE` par exemple). Les modificateurs seront donc des procédures qui seront appelées (automatiquement là encore) dès lors qu'une affectation aura lieu.

Comment s'y prendre ?

Bon, je pense pouvoir faire cela tout seul, sauf une chose : comment faire pour que ces méthodes soient appelées automatiquement ?

Il n'y a pas de technique particulière à apprendre, seulement un type particulier (ou deux) à employer. Les seuls types Ada permettant d'avoir un modificateur, un constructeur et un déestructeur par défaut appellés automatiquement, sont les types dits contrôlés (appelés `controlled`). Un second type permet de disposer d'un constructeur et d'un déestructeur automatiques (mais pas d'un modificateur), il s'agit du type `limited_controlled`.

Ces deux types ne sont pas des types standards, vous devrez donc faire appel au package `Ada.Finalization` pour les utiliser. Si vous prenez la peine d'ouvrir le fichier `a-finali.ads` vous découvrirez la déclaration de ces deux types :

Code : Ada

```
type Controlled is abstract tagged private;
type Limited_Controlled is abstract tagged limited private;
```

Eh oui ! Ces deux types sont des types privés, abstraits et donc étiquetés : le type `Limited_Controlled` est même privé et limité. Mais vous savez ce que cela signifie désormais : ces deux types sont inutilisables en l'état, il va falloir les dériver. Et si vous continuez à regarder ce package, vous trouverez les méthodes suivantes :

Code : Ada

```
procedure Initialize (Object : in out Controlled);
procedure Adjust      (Object : in out Controlled);
procedure Finalize   (Object : in out Controlled);
...
procedure Initialize (Object : in out Limited_Controlled);
procedure Finalize   (Object : in out Limited_Controlled);
```

Ce sont les fameuses méthodes dont je vous parle depuis le début de ce chapitre, méthodes qu'il nous faudra réécrire :

- `Initialize()` est notre constructeur,
- `Finalize()` est notre déestructeur,
- `Adjust()` est notre modificateur.

Mise en œuvre**Mise en place de nos types**

Bon c'est bien beau tout cela, mais j'en fais quoi, moi, de ces types contrôlés ?

Eh bien dérivons-les ! Ou plutôt implémentons-les puisque ce sont des types abstraits. Reprenons l'exemple du type `T_Heros` et étoffons-le : un héros disposera d'un niveau d'expérience. Ce qui nous donne la déclaration suivante :

Code : Ada

```
--On définit notre type "pointeur sur la classe T_Unit"
Type T_Unit_Access is access all T_Unit'class;
--On définit ensuite notre type T_Heros !
Type T_Heros is new Controlled with record
  exp : integer;
  unit : T_Unit_Access;
end record;
```

Nous obtenons ainsi un type contrôlé, c'est à dire un type étiqueté dérivant du type `controlled`. Et puisque ce type est étiqueté, nous pouvons également dériver le type `T_Heros` à volonté, par exemple en créant un nouveau type fils appelé `T_Super_Heros`. Les super-héros auront eux un niveau de mana pour utiliser de la magie ou des super-pouvoirs :

Code : Ada

```
type T_Super_Heros is new T_Heros with record
  mana : integer;
end record;
```

Ce nouveau type sera lui aussi contrôlé : comme vous devez maintenant le savoir, tout type dérivant de `T_Heros` sera lui aussi un type contrôlé.



Les types `limited controlled`, fonctionnent de la même manière. Les codes ci-dessus sont donc directement transposables aux types contrôlés et limités. Vous comprendrez donc que je ne détaillerai pas plus avant leur implémentation.

Mise en place de nos méthodes

Venons-en maintenant, à nos constructeurs, modificateurs et destructeurs. Nous ne traiterons que le cas des types `controlled`.

Constructeur

Commençons par réécrire notre constructeur `Initialize()` :

Code : Ada

```
procedure Initialize (Object : in out T_Heros) is
begin
  Object.exp := 0;
  Object.unit := null;
end Initialize;
```

Le code ci-dessus initialise notre Héros avec des valeurs nulles par défaut. Mais il est également possible de laisser la main à l'utilisateur :

Code : Ada

```
procedure Initialize (Object : in out T_Heros) is
begin
  Put_line("Quelle est le niveau d'expérience de votre héros ?");
  get(Object.exp); skip_line;
  Object.unit := choix_unit;           --on appelle ici la méthode
  d'initialisation qui demande      --à l'utilisateur de choisir
  entre plusieurs types d'unités
end Initialize;
```

Dès lors, il suffira que vous déclariez un objet de type `T_Heros` pour que cette procédure soit automatiquement appelée. La ligne ci-dessous :

Code : Ada

```
Massacror : T_Heros;
```

équivaudra à elle seule à celle-ci :

Code : Ada

```
Massacror : T_Heros := Massacror.Initialize; --c'est plus long
et surtout complètement inutile
```

Destructeur

Le destructeur sera généralement inutile. Mais dans le cas d'un type nécessitant des pointeurs, je vous recommande grandement son emploi afin de vous épargner les nombreuses désallocation de mémoire. Commençons par établir notre procédure de libération de mémoire :

Code : Ada

```
Type T_Unit_Access is access all T_Unit'class;
procedure free is new Ada.Unchecked_Deallocation(T_Unit'class,
T_Unit_Access);
```

Pour notre méthode `Finalize()`, la seule chose que nous avons à faire, c'est de libérer la mémoire pointée par l'attribut `Unit`. Cela nous donnerait ceci :

Code : Ada

```
procedure Finalize (Object : in out T_Heros) is
begin
  free(Object.unit);
end Finalize;
```

Prenons un exemple d'utilisation désormais. Déclarons un objet de type `T_Heros` soit dans une procédure-fonction soit dans un bloc de déclaration :

Code : Ada

```
... Massacror : T_Heros; --appel automatique de Initialize
begin
...
  --On effectue ce que l'on veut avec notre objet Massacror
end;
--la portée de l'objet Massacror s'arrête ici
--la méthode Finalize est appellée automatiquement
```

Modifieur

Finissons en beauté avec le modifieur. Pour bien comprendre, nous allons commencer par un exemple d'application de la méthode `Adjust()` :

Code : Ada

```
...
  Massacror, Vengeator : T_Heros ;
begin
  Massacror := Vengeator ;           -- affectation d'un objet vers
  un autre
...
```

Mais ce petit bout de code anodin présente deux soucis :

- L'attribut-pointeur `Massacror.Unit` est modifié, certes, mais qu'advient-il de l'emplacement mémoire sur lequel il pointait ? Eh bien cet emplacement n'est pas libéré et de plus, vous perdez son adresse !
- L'attribut `Massacror.unit` est un pointeur : il va donc pointer sur le même emplacement mémoire que `Vengeator.Unit` alors qu'il serait préférable qu'il pointe sur un nouvel emplacement mémoire dont le contenu serait identique à celui de `Vengeator.Unit`.

Voilà pourquoi il est important de définir des types contrôlés. Ces deux problèmes peuvent être réglés : le premier sera réglé grâce à la méthode `Finalize()`, le second va constituer le cahier des charges de notre procédure `adjust()`.



Mais comment faire une procédure d'affectation avec un seul paramètre ?

Eh bien, vous devez comprendre que la procédure `Adjust()` ne se charge pas de l'affectation en elle-même mais de l'affaiblissement de notre objet. Lors d'une affectation, la procédure `Finalize()` est appelée en tout premier (détruisant ainsi l'objet `Massacror` et résolvant le premier problème), puis l'affectation est opérée (résolvant le second problème). La méthode `Adjust()` est enfin appelée en tout dernier lieu pour rectifier les erreurs commises à l'affectation. Il nous reste donc seulement à créer un nouvel emplacement mémoire sur lequel pointera `Massacror.Unit`:

Code : Ada

```
procedure Adjust (Object : in out T_Heros) is
  Temp : T_Unit'class := Object.Unit.all ;
begin
  Object.Unit := new T_Unit'class ;
  Object.Unit.all := Temp ;
end ;
```

Voilà notre problème réglé : l'attribut-pointeur va désormais pointer sur un nouvel emplacement mémoire au contenu identique. Maintenant que nos trois méthodes sont écrites, revenons à notre exemple :

Code : Ada

```
...
  Massacror, Vengeator : T_Heros ;      -- appel automatique de
Initialize
begin
  ...
  Massacror := Vengeator ;            -- 1. appel automatique de
Finalize                         -- 2. affectation
                                    -- 3. appel automatique de
Adjust
...
end ;                                -- appel automatique de
Finalize
```

Avant d'en venir aux types `Limited_Controlled`, un dernier exemple d'application de la méthode `Adjust` avec un type `T_Horaire`. Comme dit au début de ce chapitre, un horaire ne peut valoir 0H 75Min 20s ! Définissons donc un type contrôlé `T_Horaire` et sa méthode `Adjust()`.

Code : Ada

```
type T_Horaire is new Controlled with record
  H, Min, Sec : Natural ;
end record ;

procedure Adjust (Object : in out T_Horaire) is
begin
  if Object.sec > 60
    then Object.H := Object.H + Object.sec / 3600 ;
          Object.sec := Object.sec mod 3600 ;
          Object.Min := Object.Min + Object.sec / 60 ;
          Object.sec := Object.sec mod 60 ;
  end if ;
  if Object.Min > 60
    then Object.H := Object.H + Object.min / 60 ;
          Object.min := Object.min mod 60 ;
  end if ;
end Adjust ;
```

Types contrôlés et limités

Les types `Limited_Controlled` fonctionnent exactement comme les types `Controlled`, à la différence qu'ils sont limités. Aucune affectation n'est donc possible avec ces types et la méthode `Adjust()` est donc inutile.

C'est cette fois terminé des longs chapitres de théorie sur la Programmation Orientée Objet. Ce dernier chapitre sur la Programmation Orientée Objet aura été le plus court de tous, mais comme il nécessitait des notions sur les types limités, la dérivation des types étiquetés, la programmation par classe de type... nous ne pouvions commencer par lui. Vous aurez ainsi vu les notions essentielles de la POO : encapsulation, généricité, héritage, dérivation, abstraction, polymorphisme et pour finir, finalisation. Vous allez enfin pouvoir passer à un bon vieux TP pour mettre tout cela en pratique. Après quoi, nous aborderons des thèmes aussi différents et exigeants que les exceptions, l'interface ou le multitasking.

En résumé :

- La finalisation vous permet d'automatiser l'emploi de certaines méthodes : les constructeurs pour initialiser vos objets, les destructeurs pour détruire proprement vos objets et les modificateurs qui vont normaliser vos objets après chaque affectation.
- Le constructeur est appellé automatiquement lors de la déclaration de vos objets. Le destructeur est appellé automatiquement lorsqu'un objet arrive en fin de vie (à la fin d'une sous-programme ou d'un bloc de déclaration).
- Lors d'une affectation d'un objet à un autre, Ada utilisera tout d'abord son destructeur avant d'affecter, puis il exécutera automatiquement le modificateur.
- Vous devez faire appel au package `Ada.Finalization` pour pouvoir utiliser les types contrôlés et les méthodes afférentes. Deux types sont disponibles : `Controlled` et `Limited_Controlled`. Ces types sont étiquetés, privés (voire limités) et abstraits et doivent donc être dérivés.
- Par héritage, vos types dérivés de `Controlled` et `Limited_Controlled` bénéficient des méthodes `Initialize()` (constructeur) et `Finalize()` (destructeur), voire de `Adjust()` (modificateur) pour les types non limités.

[TP] Bataille navale

Après tous ces chapitres sur la POO, nous allons prendre un peu de bon temps pour mettre toutes les notions apprises en pratique. Je vous propose de nouveau de créer un jeu. Il ne s'agira pas d'un jeu de plateau mais d'un jeu de bataille navale. Mais attention ! Pas le bon vieux jeu où on visait au hasard les cases B7 puis E4 en espérant toucher ou couler un porte-avion ! Non, Je pensais plutôt à une sorte de jeu de rôle au tour par tour. Vous disposerez d'un navire (une frégate, un galion...) et devrez affronter un navire adverse, le couler ou l'arrêter. Le but étant de gagner de l'or, de perfectionner votre navire, d'augmenter votre équipage...

L'objectif est bien sûr de mettre en œuvre la conception orientée objet dont je vous parle et repartir depuis plusieurs chapitres. Comme à chaque fois, je commencerai par vous décrire le projet, ses objectifs, ses contraintes... puis je vous fournirai quelques conseils ou pistes avant de vous livrer une solution possible. Prêt ? Alors à l'abordage ! 🚢

⚠️ Ce TP est long à réaliser. S'il exige que vous maîtrisez la POO, il n'exige pas que vous soyiez devenus un maître en matière. Toutefois, il vous faudra sûrement quelques jours pour en venir à bout. Bon courage.

Règles du jeu

Le déroulement

Comme indiqué en introduction, il s'agira d'un jeu de rôle au tour par tour. Votre navire enchaînera les combats. Entre deux combats, trois actions seront possibles :

- **Retourner au port** pour effectuer des réparations, recruter des matelots, améliorer votre navire ou sauvegarder.
- **Rester en mer** pour de nouveaux combats.
- **Partir au large** pour affronter des adversaires plus puissants.

À chaque combat, le jeu proposera plusieurs choix au joueur :

- **Bombardeur** : le navire attaqua son adversaire à coups de canon.
- **Aborder** : le navire attaqua son adversaire en lançant un abordage.
- **Défendre** : l'équipage du navire se préparera à subir un abordage et renforcera temporairement sa défense.
- **Manœuvrer** : le navire tentera d'esquiver les boulets de canon de l'adversaire.
- **Fuir** : le joueur met fin à l'affrontement et fuit.

Les navires

Il existe différents types de navires : les navires de guerre, les navires de commerce, les navires pirates et les navires corsaires. Ceux-ci pourront être classés en deux catégories :

- **Navires légaux** : cette catégorie comprendra les navires de guerre, les navires de commerce et les navires corsaires. Ils auront la capacité de bombarder les navires adverses afin de les couler.
- **Navires illégaux** : cette catégorie comprendra les navires pirates et les navires corsaires. Ils auront la capacité de se lancer à l'abordage des navires adverses afin de les capturer.

Vous aurez remarqué que les corsaires feront partie des deux catégories ce qui devrait vous rappeler l'héritage multiple. Chaque type de navire aura bien sûr ses spécificités :

- **Navires de guerre** : ils disposeront d'une force de frappe supérieure aux autres navires.
- **Navires de commerce** : les plus faibles, ils auront la capacité de vendre des marchandises dans les ports, leur permettant de gagner davantage d'or.
- **Navires pirates** : navires rapides, ils gagneront davantage d'or à l'issue d'un combat.
- **Navires corsaires** : sans capacité particulière supplémentaire.

Les statistiques

Tout navire disposera des statistiques suivantes, ainsi que des bonus associés :

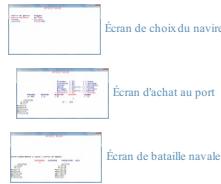
- **Coupe** : mesure l'état de la coque. Une fois à 0, le navire coule et est perdu.
- **Équipage** : mesure le nombre d'hommes sur le navire. Une fois à 0, un navire ne peut plus être dirigé et est donc perdu.
- **Puissance** : mesure la puissance de feu des canons des navires légaux.
- **Attaque** : mesure la force d'attaque de l'équipage des navires légaux.
- **Cuirasse** : mesure la résistance du navire aux boulets de canon.
- **Défense** : mesure la résistance de l'équipage aux abordages.
- **Vitesse** : mesure la vitesse du navire, ce qui lui permet plus facilement d'esquiver.

À noter que les deux premières statistiques nécessiteront une valeur maximale et une valeur courante. Les navires de commerce disposeront d'un stock de **marchandises** échangeable contre de l'or au port.

Cahier des charges

Gameplay

Venons-en désormais aux détails techniques. Encore une fois, le jeu se déroulera en mode console (oui je sais, vous avez hâte de pouvoir faire des fenêtres avec des boutons et tout, mais encore un tout petit peu de patience !). Pour égayer tout cela, il serait bon que vous fassiez appel au package `NT_Console` en apportant un peu de couleur à notre vieille console ainsi que quelques bip. Le jeu devra pouvoir se jouer au clavier à l'aide des touches `←, ↑, →, ↓, Entrée` ou `Escape`. Pensez à faire appel à la fonction `get_key` pour cela. Pour vous donner un ordre d'idée de ce que j'ai obtenu, voici quelques captures d'écran. A vrai dire, j'aurais voulu insérer quelques ASCII arts mais ça ne rendait pas très bien.



Comme vous pouvez le voir sur les captures, le joueur devra avoir accès aux statistiques de son navire à tout moment et, lors des combats, à celles de son adversaire. Il est évident que les choix proposés au joueur devront être clairs (plus de « tapez sur une touche pour voir ce qui va se passer ») et que le jeu devra retourner un message à la suite de chaque action, afin par exemple de savoir si le tir de canon a échoué ou non, combien de points ont été perdus, pourquoi votre navire a perdu (coque déchiquetée ou équipage massacré ?)... Bref, cette fois, j'exige que votre programme soit présentable.

Je vais également vous imposer de proposer la sauvegarde-chargement de vos parties. Puisque vous pourrez améliorer vos navires pour vous confronter à des adversaires toujours plus puissants, il serait dommage de ne pas proposer la possibilité de sauvegarder votre jeu.

Les calculs

Pour le calcul des dégâts liés à vos coups de canons ou de sabre, je vous laisse libre (mon programme n'est d'ailleurs pas toujours très équilibré en la matière). Voici toutefois les formules que j'ai utilisées pour ma part :

$$D_{\text{bombardement}} = \frac{(P_{\text{att}} + P_{\text{bonus}})^2}{P_{\text{att}} + P_{\text{bonus}}^{\text{att}} + P_{\text{def}} + P_{\text{bonus}}^{\text{def}}}$$

$$D_{\text{abordage}} = \frac{(A_{\text{att}} + A_{\text{bonus}})^2}{A_{\text{att}} + A_{\text{bonus}}^{\text{att}} + A_{\text{def}} + A_{\text{bonus}}^{\text{def}}} \times \frac{F_{\text{att}}^{\text{actuel}}}{F_{\text{att}}^{\text{max}}}$$

Ici, D correspond aux dégâts occasionnés, P à la puissance, A à l'attaque, E à l'équipage. att à l'attaquant et def au défenseur. Pour savoir si un navire esquive ou non vos tirs, j'ai tiré un nombre entre 0 et 100 et regardé s'il était ou non supérieur à :

$$\frac{(Y_{\text{att}} + V_{\text{att}}^{\text{bonus}})}{V_{\text{att}} + V_{\text{att}}^{\text{bonus}} + V_{\text{def}} + V_{\text{def}}^{\text{bonus}}} \times 120$$

Où V est bien entendu la vitesse du navire, 120 étant un coefficient permettant de compliquer encore davantage l'esquive des boulets. Ces formules valent ce qu'elles valent, elles ne sont pas paroles d'évangile : si vous disposez de formules plus pertinentes, n'hésitez pas à les utiliser.

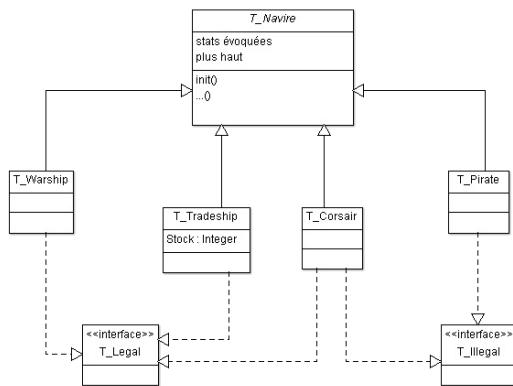
POO

Enfin, pour ceux qui ne l'auraient pas encore compris, je vais vous imposer l'emploi de «*tactiques orientées objet*» :

- utiliser des types **TAGGED** pour vos navires;
- utiliser au moins une **INTERFACE** et un type **ABSTRACT**;
- disposer de méthodes polymorphes (je n'ai pas dit que toutes devaient l'être);
- utiliser des méthodes **PRIVATE**;
- créer au moins un couple de packages père-fils.

Attention, je ne vous impose pas que tout votre code soit conçu façon objet : tout n'a pas besoin d'être encapsulé, toutes les méthodes n'ont pas besoin d'être polymorphes... Peut-être simplement que ces cinq points (au moins) soient intégrés à votre projet. Rien ne vous empêche d'employer la généricité ou des types contrôlés si le cœur vous en dit, bien entendu.

Voici le diagramme de classe que j'ai utilisé (encore une fois, rien ne vous oblige à employer le même). Libre à vous de faire hériter les méthodes de bombardement et d'abordage de vos interfaces ou de votre classe mère.



Une solution possible

L'organisation

Comme à chaque TP, je compte bien vous fournir une solution. Mais cette solution est bien entendu perfectible. Elle est également complexe et nécessitera quelques explications. Tout d'abord, évoquons l'organisation de mes packages :

- **Game** : la procédure principale. Elle ne contient que très peu de chose, l'essentiel se trouvant dans les packages.
- **P_Point** : dispose du type **T_Point** et les méthodes associées. Ce type permet pour chaque statistique (par exemple la vitesse), de disposer d'une valeur maximale, d'une valeur courante et d'une valeur bonus.
- **P_Navire** : gère les classes, méthodes et interfaces liées aux différents types navires. Ce package aura du être scindé en plusieurs sous-package pour respecter les règles d'encapsulation. Mais pour que le code gagne en lisibilité et surtout en compacité, j'ai préféré tout réunir en un seul package. Cela vous permettra d'analyser plus facilement le code, les méthodes...
- **P_Navire.list** : le package fils. Celui-ci est fait pour proposer différents navires (frégates, galions, corvette...) au joueur et à l'ordinateur.
- **P_Variabels** : gère les variables globales du projet (coût d'une amélioration, temps d'affichage des messages...) ainsi que les fonctionnalités aléatoires.
- **P_Data** : gère l'enregistrement et la sauvegarde des navires dans des fichiers texte.
- **P_Screen** : gère tous les affichages : affichage de texte accentué, affichage des messages de combat (tir manqué, abordage réussi, faite...), affichage des différents menus (menu de combat, menu du marché, menu pour choisir entre une nouvelle partie et une ancienne, menu pour choisir le type de bateau sur lequel vous souhaitez naviguer...)
- **P_Modes** : package le plus important, c'est lui qui gère tout ce que vous faites au clavier, que ce soit durant les phases de combat, lors de l'affichage du menu du marché... Bref, c'est le cœur du jeu puisqu'il gère les différents modes de jeu. Attention, les menus sont affichés grâce à **P_Screen**, mais seulement affichés, leur utilisation se fait grâce à **P_Modes**.

Le code

Game.adb

Venons-en au code à proprement parler. Voici à quoi se limite la procédure principale :

Code : Ada - game.adb

```

WITH Nt_Console ;           USE Nt_Console ;
WITH P_Navire ;             USE P_Navire ;
WITH P_Modes ;              USE P_Modes ;

PROCEDURE Game IS
BEGIN
    set_cursor(visible => false);
    DECLARE
        Joueur : T_Navire'Class := first_mode;
    BEGIN
        market_mode(joueur);
    END;
END Game;
  
```

On déclare un objet joueur de la classe **T_Navire** que l'on initialise en lançant **First_mode()**. Celui-ci proposera de choisir entre créer une nouvelle partie et en charger une ancienne. Dans chacun des cas, un Navire est renvoyé.

Ensuite, on lance le mode marché (**Market_mode()**) lequel se chargera de lancer les autres modes si besoin est.

P_Point

Ce package présente peu d'intérêt et de difficultés :

Secret (cliquez pour afficher)

Code : Ada - P_Point.ads

```

PACKAGE P_Point IS
    TYPE T_Point IS TAGGED RECORD
        Max, Current, Bonus : Integer;
    END RECORD;

    Function "-" (left : T_Point; right : integer) RETURN integer;
    FUNCTION Total (P : T_Point) RETURN Integer;
    FUNCTION Ecart (P : T_Point) RETURN Integer;
    PROCEDURE Raz (P : OUT T_Point);
  
```

```

PROCEDURE Init(P : IN OUT T_Point) ;
PROCEDURE Init(P : IN OUT T_Point ; N : Natural) ;

end P_Point ;

```

Code : Ada - P_Point.adb

```

PACKAGE BODY P_Point IS
  FUNCTION "-" (Left : T_Point ; Right : Integer) RETURN Integer
IS
BEGIN
  RETURN Integer'Max(0,Left.Current - Right) ;
END "-";
  FUNCTION Total(P : T_Point) RETURN Integer IS
BEGIN
  RETURN P.Current + P.Bonus ;
END Total ;
  FUNCTION Ecart(P : T_Point) RETURN Integer IS
BEGIN
  RETURN P.Max - P.Current ;
END Ecart ;
  PROCEDURE Raz(P : OUT T_Point) IS
BEGIN
  P.Bonus := 0 ;
END Raz ;
  PROCEDURE Init(P : IN OUT T_Point) IS
BEGIN
  P.Current := P.Max ;
  P.Bonus := 0 ;
END Init ;
  PROCEDURE Init(P : IN OUT T_Point ; N : Natural) IS
BEGIN
  P.Max := N ;
  P.Current := P.Max ;
  P.Bonus := 0 ;
END Init ;
end P_Point ;

```

P_Navire

L'un des principaux packages du jeu. Il définit mon type `T_Navire` comme **ABSTRACT**, mes interface `T_Legal` et `T_Illegal`. Trois méthodes abstraites sont proposées : `Bombarde()`, `Aborde()` et `Init()`. Des trois, seule `Init()` est réellement polymorphe. Les deux autres emploient chacune une méthode privée afin de limiter le code. J'aurais pu choisir de modifier le code de bombardement pour les navires de guerre, mais j'ai préféré modifier les statistiques pour accroître les points de Puissance initiaux afin de fournir un avantage initial aux navires de guerre qui peut toutefois être comblé pour les autres navires. Autrement dit, le bonus se fait grâce à `Init()` mais pas grâce à `Bombarde()`. Il en est de même pour `Aborde()`. Cela évite également que le code devienne gigantesque, ce qui ne vous faciliterait pas non plus son analyse.

Code : Ada - P_Navire.ads

```

WITH P_Point ;           USE P_Point ;
WITH Ada.Strings.Unbounded ; USE Ada.Strings.Unbounded ;

PACKAGE P_Navire IS
  -----
  --TYPES ET METHODES ABSTRAITS--
  TYPE T_Navire IS ABSTRACT TAGGED RECORD
    Nom      : Unbounded_String := Null_unbounded_string ;
    Coque    : T_Point ;
    Equipage : T_Point ;
    Puissance : T_Point ;
    Attaque  : T_Point ;
    Cuirasse : T_Point ;
    Defense   : T_Point ;
    Vitesse  : T_Point ;
    Playable  : Boolean := False ;
    Gold     : Natural := 0 ;
  END RECORD ;
  TYPE T_Stat_Name IS (Coque, Equipage, Puissance, Attaque,
    Cuirasse, Defense, Vitesse, Gold);
  TYPE T_Stat IS ARRAY(T_Stat_Name RANGE coque..Gold) OF Natural ;
  PROCEDURE Bombarde(Att : IN T_Navire ; Def :      in out
T_Navire'Class) IS ABSTRACT ;
  PROCEDURE Aborde (Att : IN T_Navire ; Def :      IN OUT
T_Navire'Class) IS ABSTRACT ;
  FUNCTION Init (Nom : unbounded_string ; Stat : T_Stat) RETURN
T_Navire IS ABSTRACT ;
  TYPE T_Legal IS INTERFACE ;
  TYPE T_Illegal IS INTERFACE ;
  -----
  --TYPES CONCRETS --
  TYPE T_Warship IS NEW T_Navire AND T_Legal WITH NULL RECORD ;
  TYPE T_Tradeship IS NEW T_Navire AND T_Legal WITH RECORD
    Stock : Integer ;
  END RECORD ;
  TYPE T_Corsair IS NEW T_Navire AND T_Legal AND T_Illegal WITH
NULL RECORD ;
  TYPE T_Pirate IS NEW T_Navire AND T_Illegal WITH NULL RECORD ;
  -----
  --METHODES CONCRETES--
  OVERRIDING PROCEDURE Bombarde(Att : IN T_Warship ; Def : in out
T_Navire'Class) ;
  OVERRIDING PROCEDURE Bombarde(Att : IN T_Tradeship ; Def : in out
T_Navire'Class) ;
  OVERRIDING PROCEDURE Bombarde(Att : IN T_Corsair ; Def : in out
T_Navire'Class) ;
  OVERRIDING PROCEDURE Bombarde(Att : IN T_Pirate ; Def : IN OUT
T_Navire'Class) IS NULL ;
  OVERRIDING PROCEDURE Aborde(Att : IN T_Warship ; Def : in out
T_Navire'Class) IS null ;
  OVERRIDING PROCEDURE Aborde(Att : IN T_Tradeship ; Def : in out
T_Navire'Class) IS null ;
  OVERRIDING PROCEDURE Aborde(Att : IN T_Corsair ; Def : in out
T_Navire'Class) ;
  OVERRIDING PROCEDURE Aborde(Att : IN T_Pirate ; Def : in out
T_Navire'Class) ;
  OVERRIDING FUNCTION Init (Nom : unbounded_string ; Stat :
T_Stat) RETURN T_Warship ;
  OVERRIDING FUNCTION Init (Nom : unbounded_string ; Stat :
T_Stat) RETURN T_Tradeship ;
  OVERRIDING FUNCTION Init (Nom : unbounded_string ; Stat :
T_Stat) RETURN T_Corsair ;
  OVERRIDING FUNCTION Init (Nom : unbounded_string ; Stat :
T_Stat) RETURN T_Pirate ;
  PROCEDURE Defend(Bateau : IN OUT T_Navire'Class) ;
  PROCEDURE Manoeuvre(Bateau : IN OUT T_Navire'Class) ;
  FUNCTION Est_Mort(Bateau : T_Navire'Class) RETURN boolean ;
  PROCEDURE Raz_Bonus(Bateau : OUT T_Navire'Class) ;
  PROCEDURE Repaire (Navire : in out T_Navire'Class) ;
  PROCEDURE Reparate (Navire : in out T_Navire'Class) ;
  PROCEDURE Vendre (Navire : IN OUT T_Tradeship) ;
  PROCEDURE Ameliorer_Coque (Navire : IN OUT T_Navire'Class) ;
  PROCEDURE Ameliorer_Equipage (Navire : IN OUT T_Navire'Class) ;

```

```

PROCEDURE Ameliorer_Puissance(Navire : IN OUT T_Navire'Class) ;
PROCEDURE Ameliorer_Attaque (Navire : IN OUT T_Navire'Class) ;
PROCEDURE Ameliorer_Cuirasse (Navire : IN OUT T_Navire'Class) ;
PROCEDURE Ameliorer_Defense (Navire : IN OUT T_Navire'Class) ;
PROCEDURE Ameliorer_Vitesse (Navire : IN OUT T_Navire'Class) ;

PROCEDURE Bat(Vainqueur : IN OUT T_Navire'Class ; Perdant :     IN
T_Navire'Class) ;
PROCEDURE Perd(Navire : IN T_Navire'Class) ;

PRIVATE
  PROCEDURE Private_Bombarde(Att : IN T_Navire'Class ; Def : IN OUT
T_Navire'Class) ;
  PROCEDURE Private_Aborde (Att : IN T_Navire'Class ; Def : IN OUT
T_Navire'Class) ;
  FUNCTION Esquive(Def : IN T_Navire'Class ; Att :           IN
T_Navire'Class) RETURN Boolean ;
END P_Navire ;

```

Secret (cliquez pour afficher)

Code : Ada - P_Navire.adb

```

WITH P_Variables ;          USE P_Variables ;
WITH Ada.Text_Io ;          USE Ada.Text_Io ;
WITH P_Screen ;             USE P_Screen ;
WITH NT_Console ;           USE NT_Console ;

PACKAGE BODY P_Navire IS

  -----
  --CAPACITE D'ESQUIVE AU COURS D'UN ASSAUT--
  --

  FUNCTION Esquive(Def : IN T_Navire'Class ; Att :           IN
T_Navire'Class) RETURN Boolean IS
  BEGIN
    RETURN Random > (Att.Vitesse.total*120 / (Att.Vitesse.total +
Def.Vitesse.Total)) ;
  END Esquive ;

  -----
  --CAPACITE DE BOMBARDE --
  --

  PROCEDURE Private_Bombarde(Att : IN T_Navire'Class ; Def :   in
out T_Navire'Class) IS --FORMULE GENERALE
  degats : integer := 0 ;
  BEGIN
    IF Def.Esquive(Att)
      THEN IF att.Playable
        THEN Put_Esquivé_Message(Light_Red) ;
        else Put_Esquivé_Message(Green) ;
      end if ;
    ELSE
      Degats := Integer(
        float(Att.Puissance.total**2) /
        float(Att.Puissance.Total + Def.Cuirasse.Total)) ;
      Def.Coque.Current := Def.Coque - Degats ;
      IF Att.Playable
        THEN Put_Bombard_Message(Degats,Green) ;
        else Put_Bombard_Message(Degats,Light_Red) ;
      END IF ;
    END IF ;
  END Private_Bombarde ;

  OVERRIDING PROCEDURE Bombarde(Att : IN T_Warship ; Def :   in
out T_Navire'Class) IS --POUR BATEAUX DE GUERRE
  BEGIN
    Private_Bombarde(Att,Def) ;
  END Bombarde ;
  OVERRIDING PROCEDURE Bombarde(Att : IN T_Tradeship ; Def :   in
out T_Navire'Class) IS --POUR BATEAUX COMMERCIAUX
  BEGIN
    Private_Bombarde(Att,Def) ;
  END Bombarde ;
  OVERRIDING PROCEDURE Bombarde(Att : IN T_Corsair ; Def :   in
out T_Navire'Class) IS --POUR CORSAIRES
  BEGIN
    Private_Bombarde(Att,Def) ;
  END Bombarde ;

  -----
  --CAPACITE D'ABORDER --
  --

  PROCEDURE Private_aborde(Att : IN T_Navire'Class ; Def : in out
T_Navire'Class) IS --FORMULE GENERALE
  degats : integer := 0 ;
  BEGIN
    IF Def.Esquive(Att)
      THEN IF att.Playable
        THEN Put_Esquivé_Message(Light_Red) ;
        else Put_Esquivé_Message(Green) ;
      END IF ;
    ELSE
      Degats := Integer(
        float(Att.Attaque.Total)**2 /
        float(Att.Attaque.Total + Def.Defense.Total) * 
        float(Att.Equipage.Current)/float(Att.Equipage.Max)) ;
      Def.Equipage.Current := Def.Equipage - Degats ;
      IF Att.Playable
        THEN Put_Aborde_Message(Degats,Green) ;
        else Put_Aborde_Message(Degats,Light_Red) ;
      END IF ;
    END IF ;
  END Private_aborde ;

  OVERRIDING PROCEDURE aborde(Att : IN T_Corsair ; Def : in out
T_Navire'Class) IS --POUR CORSAIRES
  BEGIN
    Private_aborde(Att,Def) ;
  END aborde ;
  OVERRIDING PROCEDURE aborde(Att : IN T_Pirate ; Def :   in out
T_Navire'Class) IS --POUR PIRATES
  BEGIN
    Private_aborde(Att,Def) ;
  END aborde ;

  -----
  --INITIALISATION-
  --

  FUNCTION Init  (Nom : unbounded_string ; Stat :           T_Stat)
RETURN T_Warship IS
  Navire : T_Warship ;
  BEGIN
    Navire.Nom := Nom ;
    Navire.Coque.init(Stat(Coque)) ;
    Navire.Equipage.init (Stat(Equipage)) ;
    Navire.Puissance.init (Stat(Puissance)*120/100) ;
    Navire.Attaque.init (Stat(Attaque)) ;
    Navire.Cuirasse.init (Stat(Cuirasse)) ;
    Navire.Defense.init (Stat(Defense)) ;
    Navire.Vitesse.init (Stat(Vitesse)*70/100) ;
    Navire.Playable := False ;
    Navire.Gold := Stat(Gold) ;
  RETURN Navire ;
  END Init ;

  FUNCTION Init  (Nom : unbounded_string ; Stat :           T_Stat)
RETURN T_Tradeship IS
  Navire : T_Tradeship ;
  BEGIN
    Navire.Nom := Nom ;
    Navire.Coque.init(Stat(Coque)) ;
    Navire.Equipage.init (Stat(Equipage)) ;
    Navire.Puissance.init (Stat(Puissance)*70/100) ;
    Navire.Attaque.init (Stat(Attaque)) ;
    Navire.Cuirasse.init (Stat(Cuirasse)*110/100) ;
    Navire.Defense.init (Stat(Defense)*110/100) ;
  
```

```

        Navire.Vitesse.init (Stat(Vitesse)*90/100) ;
        Navire.Playable := False ;
        Navire.Gold := Stat(Gold) ;
        Navire.Stock := 15 ;
        RETURN Navire ;
    END Init ;

    FUNCTION Init (Nom : unbounded_string ; Stat : T_Stat)
    RETURN T_Corsair IS
        Navire : T_Corsair ;
    BEGIN
        Navire.Nom := Nom ;
        Navire.Coque.init(Stat(Coque)) ;
        Navire.Equipage.init(Stat(Equipage)) ;
        Navire.Puissance.init(Stat(Puissance)) ;
        Navire.Attaque.init (Stat(Attaque)) ;
        Navire.Cuirasse.init (Stat(Cuirasse)) ;
        Navire.Defense.init (Stat(Defense)) ;
        Navire.Vitesse.init (Stat(Vitesse)) ;
        Navire.Playable := False ;
        Navire.Gold := Stat(Gold) ;
        RETURN Navire ;
    END Init ;

    FUNCTION Init (Nom : unbounded_string ; Stat : T_Stat)
    RETURN T_Pirate IS
        Navire : T_Pirate ;
    BEGIN
        Navire.Nom := Nom ;
        Navire.Coque.init(Stat(Coque)) ;
        Navire.Equipage.init(Stat(Equipage)) ;
        Navire.Puissance.init(Stat(Puissance)) ;
        Navire.Attaque.init (Stat(Attaque)*120/100) ;
        Navire.Cuirasse.init (Stat(Cuirasse)*70/100) ;
        Navire.Defense.init (Stat(Defense)) ;
        Navire.Vitesse.init (Stat(Vitesse)*130/100) ;
        Navire.Playable := False ;
        Navire.Gold := Stat(Gold) ;
        RETURN Navire ;
    END Init ;

    -----
    --CAPACITÉ DE DEFENSE --
    ----

    PROCEDURE Defend(Bateau : IN OUT T_Navire'Class) IS
    BEGIN
        Bateau.Defense.Bonus := Bateau.Defense.Current * 25 /100 ;
        IF Bateau.Playable
            THEN Put_Defense_Message(Green) ;
        ELSE Put_Defense_Message(Light_Red) ;
        END IF ;
    END Defend ;

    -----
    --CAPACITÉ DE MANOEUVRE--
    ----

    PROCEDURE Manoeuvre(Bateau : IN OUT T_Navire'Class) IS
    BEGIN
        Bateau.Vitesse.Bonus := Bateau.Vitesse.Current * 25 /100 ;
        IF Bateau.Playable
            THEN Put_Maneuvre_Message(Green) ;
        ELSE Put_Maneuvre_Message(Light_Red) ;
        END IF ;
    END Manoeuvre ;

    -----
    --FONCTION POUR SAVOIR SI UN BATEAU EST COULE --
    ----

    FUNCTION Est_Mort(Bateau : T_Navire'Class) RETURN Boolean IS
    BEGIN
        RETURN Bateau.Coque.current = 0 OR Bateau.Equipage.current =
    0 ;
    END Est_Mort ;

    -----
    --REMISE À ZÉRO POUR DBT DE TOUR --
    ----

    PROCEDURE Raz_Bonus(Bateau : OUT T_Navire'Class) IS
    BEGIN
        Bateau.Coque.raz ;
        Bateau.Equipage.raz ;
        Bateau.Puissance.raz ;
        Bateau.Attaque.raz ;
        Bateau.Cuirasse.raz ;
        Bateau.Defense.raz ;
        Bateau.Vitesse.raz ;
    END Raz_Bonus ;

    -----
    --MÉTHODES POUR LE MARCHÉ --
    ----

    PROCEDURE Reparer (Navire : IN OUT T_Navire'Class) IS
        Cout : constant Natural := (Navire.Coque.Max -
        Navire.Coque.Current) * Repair_Cost ;
    BEGIN
        IF Cout <= Navire.Gold
            THEN Navire.Coque.Init ;
            Navire.Gold := Navire.Gold - Cout ;
        ELSE Navire.Coque.Current := Navire.Coque.Current +
        (Navire.Gold / Repair_Cost) ;
            Navire.Gold := Navire.Gold mod Repair_Cost ;
        END IF ;
        Sleep ;
    END Reparer ;

    PROCEDURE Recruter (Navire : IN OUT T_Navire'Class) IS
        Cout : constant Natural := (Navire.Equipage.max -
        Navire.Equipage.current) * Recrute_cost ;
    BEGIN
        IF Cout <= Navire.Gold
            THEN Navire.Equipage.Init ;
            Navire.Gold := Navire.Gold - Cout ;
        ELSE Navire.Equipage.Current :=
        Navire.Equipage.Current + (Navire.Gold / Recrute_Cost) ;
            Navire.Gold := Navire.Gold mod Recrute_Cost ;
        END IF ;
        Sleep ;
    END Recruter ;

    PROCEDURE Vendre (Navire : IN OUT T_Tradeship) IS
    BEGIN
        Navire.Gold := Navire.Gold + Navire.Stock * Goods_Cost ;
        Navire.Stock := 0 ;
        Sleep ;
    END Vendre ;

    PROCEDURE Ameliorer_Coque (Navire : IN OUT T_Navire'Class)
    IS
    BEGIN
        IF Coque_Cost <= Navire.Gold
            THEN Navire.Gold := Navire.Gold - Coque_Cost ;
            Navire.Coque.Max := Navire.Coque.Max + 2 ;
            Navire.Coque.Current := Navire.Coque.Current + 2 ;
        END IF ;
        Coque_Cost := Coque_cost * 2 ;
        Sleep ;
    END Ameliorer_Coque ;

    PROCEDURE Ameliorer_Equipage (Navire : IN OUT T_Navire'Class)
    IS
    BEGIN
        IF Equipage_Cost <= Navire.Gold
            THEN Navire.Gold := Navire.Gold - Equipage_Cost ;
            Navire.Equipage.Max := Navire.Equipage.Max + 2 ;
            Navire.Equipage.Current :=
        Navire.Equipage.Current + 2 ;
        END IF ;
        Equipage_Cost := Equipage_cost * 2 ;
        Sleep ;
    END Ameliorer_Equipage ;

```

```

PROCEDURE Ameliorer_Puissance(Navire : IN OUT T_Navire'Class)
IS
BEGIN
  IF Puissance_Cost <= Navire.Gold
    THEN Navire.Gold := Navire.Gold - Puissance_Cost ;
    Navire.Puissance.Max := Navire.Puissance.Max + 1
  ;
  Navire.Puissance.Current := 
  Navire.Puissance.Current + 1 ;
  Puissance_Cost := Puissance_cost * 2 ;
END IF ;
Sleep ;
END Ameliorer_Puissance ;

PROCEDURE Ameliorer_Attaque (Navire : IN OUT T_Navire'Class)
IS
  cout : constant natural := Attaque_Cost *
Navire.Equipage.max ;
BEGIN
  IF cout <= Navire.Gold
    THEN Navire.Gold := Navire.Gold - cout ;
    Navire.Attaque.Max := Navire.Attaque.Max + 1 ;
    Navire.Attaque.Current := Navire.Attaque.Current
+ 1 ;
    Attaque_Cost := Attaque_cost + 1 ;
END IF ;
Sleep ;
END Ameliorer_Attaque ;

PROCEDURE Ameliorer_Cuirasse (Navire : IN OUT T_Navire'Class)
IS
BEGIN
  IF Cuirasse_Cost <= Navire.Gold
    THEN Navire.Gold := Navire.Gold - Cuirasse_Cost ;
    Navire.Cuirasse.Max := Navire.Cuirasse.Max + 1 ;
    Navire.Cuirasse.Current :=
  Navire.Cuirasse.Current + 1 ;
    Cuirasse_Cost := Cuirasse_cost * 2 ;
END IF ;
Sleep ;
END Ameliorer_Cuirasse ;

PROCEDURE Ameliorer_Defense (Navire : IN OUT T_Navire'Class)
IS
  cout : constant natural := Defense_Cost *
Navire.Equipage.max ;
BEGIN
  IF cout <= Navire.Gold
    THEN Navire.Gold := Navire.Gold - cout ;
    Navire.Defense.Max := Navire.Defense.Max + 1 ;
    Navire.Defense.Current := Navire.Defense.Current
+ 1 ;
    Defense_Cost := Defense_cost + 1 ;
END IF ;
Sleep ;
END Ameliorer_Defense ;

PROCEDURE Ameliorer_Vitesse (Navire : IN OUT T_Navire'Class)
IS
BEGIN
  IF Vitesse_Cost <= Navire.Gold
    THEN Navire.Gold := Navire.Gold - Vitesse_Cost ;
    Navire.Vitesse.Max := Navire.Vitesse.Max + 1 ;
    Navire.Vitesse.Current := Navire.Vitesse.Current
+ 1 ;
    Vitesse_Cost := Vitesse_cost * 2 ;
END IF ;
Sleep ;
END Ameliorer_Vitesse ;

--GAIN DE FIN DE COMBAT--

PROCEDURE Bat(Vainqueur : IN OUT T_Navire'Class ; Perdant : IN
T_Navire'Class) IS
  Gain_or, Gain_Stock : Natural := 0 ;
BEGIN
  goto_xy(2,10) ;
  set_foreground(green) ;
  Put("VOUS AVEZ VAINCU VOTRE ADVERSAIRE !") ;
  IF Vainqueur IN T_Pirate'Class
    THEN Gain_Or := Perdant.Gold / 4 ;
    ELSE Gain_Or := Perdant.Gold / 5 ;
  END IF ;
  Vainqueur.Gold := Vainqueur.Gold + Gain_Or ;
  goto_xy(0,11) ;
  set_foreground(green) ;
  Put("Vous remportez " & integer'image(Gain_or) & " or") ;
  IF Vainqueur IN T_Tradeship'Class
    THEN Gain_stock := (Niveau-1) * 3 + 1 ;
    T_Tradeship(Vainqueur).Stock := Gain_Stock ;
  END IF ;
  Put(" et " & integer'image(Gain_Stock) & " stocks") ;
  END IF ;
  put(" !") ;
  Set_Foreground(Black) ;
  delay Message_time * 2.0 ;
END Bat ;

PROCEDURE Perd(Navire : IN T_Navire'Class) IS
BEGIN
  goto_xy(2,10) ;
  set_foreground(right_red) ;
  Put("VOUS AVEZ PERDU CONTRE VOTRE ADVERSAIRE !") ;
  goto_xy(0,11) ;
  IF Navire.Coque.Current = 0
    THEN Put("La coque de votre navire est d" &
Character'Val(130) & "truite.") ;
    ELSIF Navire.Equipage.Current = 0
      then Put("Votre " & Character'Val(130) & "quipage     est
massacr" & Character'Val(130) & ".") ;
    END IF ;
    Set_Foreground(Black) ;
    delay Message_time * 2.0 ;
  END Perd ;

END P_Navire ;

```

P_Navire.list

Peu d'intérêt, c'est ici que sont écrits les statistiques des différents navires (frégate, trois-mâts, galion et corvette). Vous pouvez en ajouter autant que bon vous semble. Le seul inconvénient, c'est qu'il faudra recompiler le code source (les données sont hard-coded).

Secret (cliquez pour afficher)

Code : Ada : p_navire-list.adb

```

package P_Navire.List is

  TYPE T_Navire_Id IS RECORD
    Nom : Unbounded_String ;
    Stat : T_Stat ;
  END RECORD ;

  type T_Liste_Navire is array(positive range <>) of T_Navire_Id
  ;

  -----
  -- NAVIRES --
  -----

  --EXAMPLE STATS Coque Equpg Puiss Att Cuir Defse
  Vits_Gold
  Stat_Fregate   : T_Stat := (15,   10,   14,    8,    8,    6,
  200) ;
  Stat_Gallion   : T_Stat := (20,   14,   10,   12,    8,
  500) ;
  Stat_Corvette  : T_Stat := (12,    7,    8,    10,    4,
  150) ;
  Stat_Trois_Mats : T_Stat := (16,    8,   12,    7,   10,    8,

```

```

10,   300) ;
Liste_Navire : T_Liste_Navire :=(
  (To_Unbounded_String("Frégate"), Stat_Fregate),
  (To_Unbounded_String("Gallion"), Stat_Gallion),
  (To_Unbounded_String("Corvette"), Stat_Corvette),
  (To_Unbounded_String("Trois-Mâts"), Stat_Trois_Mats)
);
-----
-- GENERATION D'ADVERSAIRES --
-----

PROCEDURE Mettre_Niveau(Navire : IN OUT T_Navire'Class ; Niveau
: Positive) IS
  FUNCTION Generer_Ennemi(Niveau : Natural := 1) RETURN
T_Navire'Class ;
End P_Navire.List ;

```

Code : Ada - p_navire-list.adb

```

WITH P_Variables ;           USE P_Variables ;
package body P_Navire.List is
  PROCEDURE Mettre_Niveau(Navire : IN OUT T_Navire'Class ; Niveau
: Positive) IS
    Coef : float ;
    BEGIN
      Coef := 1.5*(Niveau-1) ;
      Navire.Coque.Init(Integer(Float(Navire.Coque.Max) * Coef)) ;
      Navire.Equipage.Init(Integer(Float(Navire.Equipage.Max) *
Coef)) ;
      Navire.Puissance.Init(Integer(Float(Navire.Puissance.Max) *
Coef)) ;
      Navire.Attaque.Init(Integer(Float(Navire.Attaque.Max) *
Coef)) ;
      Navire.Cuirasse.Init(Integer(Float(Navire.Cuirasse.Max) *
Coef)) ;
      Navire.Defense.Init(Integer(Float(Navire.Defense.Max) *
Coef)) ;
      Navire.Vitesse.Init(Integer(Float(Navire.Vitesse.Max) *
Coef)) ;
      Coef := 2.0*(Niveau-1) ;
      Navire.Gold := integer(float(Navire.Gold) * Coef) ;
    END Mettre_Niveau ;

  FUNCTION Generer_Ennemi(Niveau : Natural := 1) RETURN
T_Navire'Class IS
  Navire : ACCESS T_Navire'Class ;
  N : Natural ;
  BEGIN
    N:= Random mod Liste_Navire'Length + 1 ;
    CASE Random mod 4 IS
      WHEN 0 => Navire := new
T_Warship'(init(Liste_Navire(N).nom, Liste_Navire(N).stat)) ;
      WHEN 1 => Navire := new
T_Tradeship'(init(Liste_Navire(N).nom, Liste_Navire(N).stat)) ;
      WHEN 2 => Navire := new
T_Pirate'(init(Liste_Navire(N).nom, Liste_Navire(N).stat)) ;
      WHEN others => Navire := new
T_Corsair'(init(Liste_Navire(N).nom, Liste_Navire(N).stat)) ;
    END CASE ;
    Mettre_Niveau(Navire.all,Niveau) ;
    RETURN Navire.all ;
  END Generer_Ennemi ;
End P_Navire.List ;

```

P_Variables

Là encore peu d'intérêt, hormis si vous souhaitez modifier le temps d'affichage des message, le coût de base d'une amélioration ou des réparations.

Secret (cliquez pour afficher)

Code : Ada - P_Variables.ads

```

WITH Ada.Strings.Unbounded ;           USE Ada.Strings.Unbounded ;
WITH Ada.Numerics.Discrete_Random ;

PACKAGE P_Variables IS
  subtype T_Pourcentage is integer range 0..100 ;
  PACKAGE P_Random IS
    NEW
Ada.Numerics.Discrete_Random(T_Pourcentage) ;
  Germe : P_Random.Generator ;
  PROCEDURE Reset ;
  FUNCTION Random RETURN T_Pourcentage ;
  Message_Time : CONSTANT Duration := 2.0 ;
  Repair_Cost : CONSTANT Natural := 5 ;
  Recette_Cost : CONSTANT Natural := 5 ;
  Goods_cost : CONSTANT Natural := 5 ;
  Coque_Cost : Natural := 25 ;
  Equipage_Cost : Natural := 25 ;
  Puissance_Cost : Natural := 25 ;
  Attaque_Cost : Natural := 2 ;
  Cuirasse_Cost : Natural := 25 ;
  Defense_Cost : Natural := 2 ;
  Vitesse_Cost : Natural := 25 ;
  Niveau : Natural := 1 ;
  Save_File_Name : unbounded_string := Null_unbounded_string ;

```

PRIVATE

```

  PROCEDURE Reset(G : P_Random.Generator) RENAMES P_Random.Reset
;
  FUNCTION Random(G : P_Random.Generator) RETURN T_Pourcentage
RENAMES P_Random.Random ;
end P_Variables ;

```

Code : Ada - P_Variables.adb

```

PACKAGE body  P_Variables IS
  PROCEDURE Reset IS
  BEGIN
    Reset(Germe) ;
  END Reset ;
  FUNCTION Random RETURN T_Pourcentage IS
  BEGIN
    RETURN Random(Germe) ;
  END Random ;
end P_Variables ;

```

P_Data

Comme dit précédemment, ce package sert à l'enregistrement et au chargement de parties enregistrées dans des fichiers texte. Bref, c'est du déjà-vu pour vous :

Secret (cliquez pour afficher)

Code : Ada - P_Data.ads

```

WITH P_Navire ; USE P_Navire ;
PACKAGE P_Data IS
  FUNCTION Load_Navire(File_Name : String) RETURN T_Navire'Class
  ; PROCEDURE Save_Navire(Navire : in T_Navire'Class) ;
end P_Data ;

Code : Ada - P_Data.adb

WITH Ada.Text_IO ; USE Ada.Text_IO ;
WITH Ada.Strings.Unbounded ; USE Ada.Strings.Unbounded ;
WITH P_Variables ; USE P_Variables ;
WITH P_Point ; USE P_Point ;

PACKAGE BODY P_Data IS
  FUNCTION Load_Navire(File_Name : String) RETURN T_Navire'Class
  IS
    FUNCTION Value(F : File_Type) RETURN Integer IS
      BEGIN
        RETURN Integer'Value(Get_Line(F)) ;
      END Value ;
      Navire : ACCESS T_Navire'Class ;
      F : File_Type ;
    BEGIN
      Open(F, In_File, "./data/" & File_Name & ".txt") ;
      CASE Value(F) IS
        WHEN 1 => Navire := NEW T_Warship ;
        WHEN 2 => Navire := NEW T_Tradeship ;
        WHEN 3 => Navire := NEW T_Pirate ;
        WHEN others => Navire := NEW T_Corsair ;
      END CASE ;
      Navire.Coupe := to_unbounded_string(get_line(F)) ;
      Navire.Coupe := (Value(F),Value(F),0) ;
      Navire.Equipage := (Value(F),Value(F),0) ;
      Navire.Puissance := (Value(F),Value(F),0) ;
      Navire.Attaque := (Value(F),Value(F),0) ;
      Navire.Cuirasse := (Value(F),Value(F),0) ;
      Navire.Defense := (Value(F),Value(F),0) ;
      Navire.Vitesse := (Value(F),Value(F),0) ;
      Navire.Gold := Value(F) ;
      IF Navire.All IN T_Tradeship'Class
        THEN T_Tradeship(Navire.All).Stock := Value(F) ;
      END IF ;
      Coque_Cost := value(F) ;
      Equipage_Cost := value(F) ;
      Puissance_Cost := value(F) ;
      Attaque_Cost := value(F) ;
      Cuirasse_Cost := value(F) ;
      Defense_Cost := value(F) ;
      Vitesse_Cost := value(F) ;
      Close(F) ;
      RETURN Navire.All ;
    END Load_Navire ;

PROCEDURE Save_Navire(Navire : in T_Navire'Class) IS
  F : File_Type ;
  PROCEDURE Save_Stat(F : File_Type ; P : T_Point) IS
  BEGIN
    Put_Line(F, Integer'Image(P.Max)) ;
    Put_Line(F, Integer'Image(P.Current)) ;
  END Save_Stat ;

BEGIN
  Create(F,Out_File,"./data/" & to_string(Save_File_Name) &
".txt") ;
  IF Navire IN T_Warship'Class
    THEN Put_Line(F,"1") ;
  ELSIF Navire IN T_Tradeship'Class
    THEN Put_Line(F,"2") ;
  ELSIN Navire IN T_Pirate'Class
    THEN Put_Line(F,"3") ;
  ELSIN Navire IN T_Corsair'Class
    THEN Put_Line(F,"4") ;
  END IF ;
  Put_Line (F, To_String(Navire.Nom)) ;
  Save_Stat(F, Navire.Coupe) ;
  Save_Stat(F, Navire.Puissance) ;
  Save_Stat(F, Navire.Attaque) ;
  Save_Stat(F, Navire.Cuirasse) ;
  Save_Stat(F, Navire.Defense) ;
  Save_Stat(F, Navire.Vitesse) ;
  Put_Line (F, Integer'Image(Navire.Gold)) ;
  IF Navire IN T_Tradeship'Class
    THEN Put_Line(F,
Integer'Image(T_Tradeship(Navire).Stock)) ;
  END IF ;
  Put_Line (F, Integer'Image(Coque_Cost)) ;
  Put_Line (F, Integer'Image(Equipage_Cost)) ;
  Put_Line (F, Integer'Image(Puissance_Cost)) ;
  Put_Line (F, Integer'Image(Attaque_Cost)) ;
  Put_Line (F, Integer'Image(Cuirasse_Cost)) ;
  Put_Line (F, Integer'Image(Defense_Cost)) ;
  Put_Line (F, Integer'Image(Vitesse_Cost)) ;
  Close(F) ;
  END Save_Navire ;
END P_Data ;

```

P_Screen

Package important : c'est là que se font tous les affichages.

```

Code : Ada - P_Screen.ads

WITH Nt_Console ; USE Nt_Console ;
WITH P_Navire ; USE P_Navire ;
WITH P_Point ; USE P_Point ;

PACKAGE P_Screen IS
  TYPE T_Position IS (Gauche, Droite) ;
  PROCEDURE Put(N : Integer ; Color : Color_Type := Black ; X,Y :
  Integer := -1) ;
  PROCEDURE Print(Text : String ; Color : Color_Type := Black ; X,Y :
  Integer := -1) ;
  PROCEDURE Put_Title ;
  PROCEDURE Put_First_Menu(Indice : Natural := 1) ;
  PROCEDURE Put_Fight_Menu(Navire : T_Navire'Class ; Indice :
  Natural := 1) ;
  PROCEDURE Put_NextFight_Menu(Indice : Natural := 1) ;
  PROCEDURE Put_Status(Navire : T_Navire'Class ; Pos : T_Position
  := Gauche) ;
  PROCEDURE Put_Point(Point : T_Point) ;
  PROCEDURE Put_Esquivre_Message(Color : Color_Type := Green) ;
  PROCEDURE Put_Bombard_Message(Degats : Integer ; Color :
  Color_Type := Green) ;
  PROCEDURE Put_Abdordage_Message(Degats : Integer ; Color :
  Color_Type := Green) ;
  PROCEDURE Put_Defense_Message(Color : Color_Type := Green) ;
  PROCEDURE Put_Maneuvre_Message(color : color_type := green) ;
  PROCEDURE Put_Fuite_Message ;
  PROCEDURE Put_Gold(Navire : T_Navire'Class ; X : X_Pos := 40 ; Y
  : Y_Pos := 16) ;
  PROCEDURE Put_Market_Menu(Navire : T_Navire'Class ; Indice :
  Natural := 1) ;
  PROCEDURE Put_Market_Menu2(Indice : Natural := 1) ;
  PROCEDURE Put_Select_Menu(Indice : Natural := 1) ;
  PROCEDURE Put_Select_Menu2(Indice : Natural := 1) ;

```

```
end P_Screen ;
```

Secret (cliquez pour afficher)

Code : Ada - P_Screen.adb

```
WITH Ada.Text_IO ;           USE Ada.Text_IO ;
WITH P_Variables ;          USE P_Variables ;
with P_Navire.list ;         use P_Navire.list ;
WITH Ada.Strings.Unbounded ; USE Ada.Strings.Unbounded ;
WITH Ada.Integer_Text_IO ;
```

```
PACKAGE BODY P_Screen IS
```

```
--AFFICHER UN ENTIER AVEC UNE COULEUR ET UN LIEU SPECIFIÉ--
```

```
PROCEDURE Put(N : Integer ; Color : Color_Type := black ; X,Y : Integer := -1) IS
BEGIN
  IF X >= 0 AND Y >= 0
    THEN Goto_Xy(X,Y) ;
  END IF ;
  Set_Foreground(Color) ; Set_BackGround(White) ;
  Ada.Integer_Text_IO.put(n,0)
  Set_Foreground(Black) ;
END put;
```

```
--AFFICHER UN TEXTE AVEC UNE COULEUR ET UN LIEU SPECIFIÉ--
```

```
PROCEDURE Print(Text : String ; Color : Color_Type := Black ;
X,Y : Integer := -1) IS
BEGIN
  IF X >= 0 AND Y >= 0
    THEN Goto_Xy(X,Y) ;
  END IF ;
  Set_Foreground(Color) ; Set_BackGround(White) ;
  FOR I IN Text'RANGE LOOP
    CASE Text(I) IS
      WHEN 'é' => Put(Character'Val(130)) ;
      WHEN 'â' => Put(Character'Val(131)) ;
      WHEN 'ê' => Put(Character'Val(138)) ;
      WHEN OTHERS => Put(Text(I)) ;
    END CASE ;
  END LOOP ;
  Set_Foreground(Black) ;
END Print;
```

```
--AFFICHAGE DU TITRE--
```

```
PROCEDURE Put_Title IS
BEGIN
  goto_xy(0,0) ;
  set_Background(white) ;
  Set_Foreground(light_red) ;
  Put_Line(" BATAILLE NAVALE") ; new_line ;
  Set_Foreground(black) ;
END Put_Title;
```

```
--AFFICHAGE DU PREMIER MENU--
```

```
PROCEDURE Put_First_Menu(Indice : Natural := 1) IS
  X : constant Natural := 9 ;
  Y : constant Natural := 4 ;
  color : color_type := blue ;
BEGIN
  Print("Que souhaitez-vous faire ?",color,0,Y-1) ;
  FOR I IN 1..2 LOOP
    IF Indice=I
      THEN Color := Light_Red ;
      ELSE Color := Blue ;
    END IF ;
    CASE I IS
      WHEN 1 => Print("NOUVELLE PARTIE",Color,X,Y) ;
      WHEN 2 => Print("CHARGER      UNE      ANCIENNE
PARTIE",Color,X+20,Y) ;
    END CASE ;
  END LOOP ;
  Goto_Xy(X+20*(Indice-1) + 10,Y+1) ;
  Set_Foreground(Light_Red) ;
  Put(Character'Val(30)) ;
  Set_Foreground(Black) ;
END Put_First_Menu;
```

```
--MENU DE BATAILLE--
```

```
PROCEDURE Put_Fight_Menu(Navire : T_Navire'Class ; Indice :
Natural:=1) IS
  X : constant Natural := 9 ;
  Y : constant Natural := 13 ;
  blank_line : constant string(1..80) := (others => ' ') ;
BEGIN
  Goto_Xy(0,Y) ;
  put(blank_line) ;
  Goto_Xy(X,Y) ;
  IF Indice = 1 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  IF Navire IN T_Illegal'Class THEN Put("ABORDER ") ; END IF ;
  IF Indice = 2 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  IF Navire IN T_Legal'Class THEN Put("BOMBARDER ") ; END IF ;
  IF Indice = 3 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  Put("DEFENDRE ") ;
  IF Indice = 4 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  Put("MANOEUVRER ") ;
  IF Indice = 5 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  Put("FUIR") ;
  Goto_Xy(0,Y+1) ;
  put(blank_line) ;
  Goto_Xy(X+20*(Indice-1),Y+1) ;
  Set_Foreground(Light_Red) ;
  put(character'val(30)) ;
  Set_Foreground(Black) ;
END Put_Fight_Menu;
```

```
--MENU D'APRÈS BATAILLE--
```

```
PROCEDURE Put_NextFight_Menu(Indice : Natural := 1) IS
  X : constant Natural := 9 ;
  Y : constant Natural := 12 ;
  color : color_type := blue ;
  blank_line : constant string(1..80) := (others => ' ') ;
BEGIN
  FOR I IN -2..+2 LOOP
    Goto_Xy(0,Y+1) ; Put(Blank_Line) ;
  END LOOP ;
  Print("Que souhaitez-vous faire désormais ?",color,0,Y) ;
  FOR I IN 1..3 LOOP
    IF Indice=I
      THEN Color := Light_Red ;
      ELSE Color := Blue ;
    END IF ;
    CASE I IS
      WHEN 1 => Print("DEBARQUER",Color,X,Y+1) ;
      WHEN 2 => Print("FAIRE VOILE",Color,X+20,Y+1) ;
      WHEN 3 => Print("PRENDRE LE LARGE",Color,X+40,Y+1) ;
    END CASE ;
  END LOOP ;
```

```

    END LOOP ;
    Goto_Xy(X+20*(Indice-1) + 3,Y+2) ;
    Set_Foreground(Light_Red) ;
    Put(Character'Val(30)) ;
    set_foreground(black) ;
  END Put_Nextfight_Menu ;

-----AFICHAGE DU STATUS D'UN BATEAU-----

PROCEDURE Put_Status(Navire : T_Navire'class ; Pos : T_Position
:= Gauche) IS
  X : Natural ;
  X : constant natural := 15 ;
BEGIN
  IF Pos = Gauche
    THEN X:= 0 ;
  ELSE X:= 39 ;
  END IF ;

  set_background(white) ;
  print(" " & to_string(Navire.nom), black, x, y) ;
  Set_Foreground(Light_Red) ; Goto_Xy(X,Y+1) ;
  Put(" " & Character'Val(2) & " ") ; put_point(Navire.coque) ;
;
  Set_Foreground(Green) ; Goto_Xy(X,Y+2) ;
  Put(" " & Character'Val(3) & " ") ;
  Put_Point(Navire.Equipage) ;
  Goto_Xy(X,Y+3) ; Put("Psce:") ; Put_Point(Navire.Puissance)
;
  Goto_Xy(X,Y+4) ; Put("Atq:") ; Put_Point(Navire.Attaque) ;
  Goto_Xy(X,Y+5) ; Put("Cuir:") ; Put_Point(Navire.Cuirasse) ;
  Goto_Xy(X,Y+6) ; Put("Dfns:") ; Put_Point(Navire.Defense) ;
  Goto_Xy(X,Y+7) ; Put("Vits:") ; Put_Point(Navire.Vitesse) ;
END Put_Status ;

-----AFICHAGE DU TYPE X/Y-----

PROCEDURE Put_Point(Point : T_Point) IS
BEGIN
  IF Point.Current <= Point.Max / 4
    THEN Put(Point.Current,Light_Red) ;
  ELSE Put(Point.Current) ;
  END IF ;
  Put('/') ; Put(Point.Max) ;
  IF Point.Bonus > 0
    THEN Put('+') ; Put(Point.Bonus) ;
  END IF ;
END Put_Point ;

-----AFICHAGE DES MESSAGES DU JEU-----

PROCEDURE Put_Esquivre_Message(color : color_type := green) IS
BEGIN
  goto_xy(0,11) ;
  IF Color = Green
    THEN set_foreground(green) ;
    Put("Vous avez esquivé character'val(130) &" l'attaque
adverse !");
    Set_Foreground(Black) ;
  ELSE Set_Foreground(light_Red) ;
    Put("L'adversaire a esquivé character'val(130) &" votre
attaque !");
    Set_Foreground(Black) ;
  END IF ;
  delay message_time ;
END Put_Esquivre_Message ;

PROCEDURE Put_Bombard_Message(Degats : Integer ; Color :
Color_Type := Green) IS
BEGIN
  Goto_Xy(0,11) ;
  set_foreground(black) ;
  IF Color = Green
    THEN Put("Votre bombardement a causé"
Character'Val(130) & " ");
    Put(degats,green) ;
    put(" points de dégâts") ;
    Character'Val(131) &"ts") ;
  ELSE Put("Le bombardement ennemi a causé character'Val(130)
& ") ;
    Put(degats,light_red) ;
    put(" points de dégâts") ;
    Character'Val(131) &"ts") ;
  END IF ;
  DELAY message_time ;
END Put_Bombard_Message ;

PROCEDURE Put_Abdorage_Message(Degats : Integer ; Color :
Color_Type := Green) IS
BEGIN
  Goto_Xy(0,11) ;
  set_foreground(black) ;
  IF Color = Green
    THEN Put("Votre abordage a tué character'Val(130) & ")
  ; Put(degats,green) ;
    put(" marins ennemis") ;
  ELSE Put("L'abordage ennemi a tué character'Val(130) & ")
  ; Put(degats,light_red) ;
    put(" de vos marins") ;
  END IF ;
  DELAY message_time ;
END Put_Abdorage_Message ;

PROCEDURE Put_Defense_Message(color : color_type := green) IS
BEGIN
  goto_xy(0,11) ;
  IF Color = Green
    THEN set_foreground(green) ;
    Put("Nos hommes se sont battus avec l'adversaire");
    character'Val(133) &" l'abordage !");
    Set_Foreground(Black) ;
  ELSE Set_Foreground(light_Red) ;
    Put("L'ennemi se bat avec character'Val(130) & pare
    character'Val(133) &" l'abordage !");
    Set_Foreground(Black) ;
  END IF ;
  delay Message_time ;
END Put_Defense_Message ;

PROCEDURE Put_Maneuvre_Message(color : color_type := green) IS
BEGIN
  goto_xy(0,11) ;
  IF Color = Green
    THEN set_foreground(green) ;
    Put("Votre navire se place sous le vent !");
    Set_Foreground(Black) ;
  ELSE Set_Foreground(light_Red) ;
    Put("Le navire ennemi se place sous le vent !");
    Set_Foreground(Black) ;
  END IF ;
  delay Message_time ;
END Put_Maneuvre_Message ;

PROCEDURE Put_Fuite_Message IS
BEGIN
  goto_xy(0,11) ;
  Set_Foreground(light_Red) ;
  Put("Vous fuyez devant l'adversaire" & character'Val(130) & "
!");
  Set_Foreground(Black) ;
  delay Message_time ;
END Put_Fuite_Message ;

-----AFICHAGE DES MENUS DU MARCHÉ-----

```

```

PROCEDURE Put_Gold(Navire : T_Navire'Class ; X : _X_Pos := 40 ;
Y : Y_Pos := 16) IS
  blank_line : constant string(1..80 - X) := (others => ' ') ;
BEGIN
  Goto_Xy(X,Y) ;
  Put(Blank_Line) ;
  Goto_Xy(X,Y) ;
  set_foreground(black) ;
  Put("O : ") ;
  IF Navire.Gold = 0
    THEN Set_Foreground(Light_Red) ;
  END IF ;
  Put(Integer'image(Navire.Gold)) ;
  Set_Foreground(Black) ;
END Put_Gold ;

PROCEDURE Put_Market_Menu1(Indice : Natural := 1) IS
  X : constant Natural := 9 ;
  Y : constant Natural := 13 ;
  blank_line : constant string(1..80) := (others => ' ') ;
BEGIN
  Goto_Xy(0,Y-1) ;
  Put(Blank_Line) ;
  Goto_Xy(0,Y) ;
  put(blank_line) ;

  IF Indice = 1 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  Goto_Xy(X,Y-1) ; Put("PRENDRE") ;
  Goto_Xy(X+12,Y) ; Put("(" &
integer'image((Navire.coque.max - Navire.coque.current) *
Repair_cost) &")") ;

  IF Indice = 2 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  Goto_Xy(X+12,Y-1) ; Put("REPARER") ;
  Goto_Xy(X+12,Y) ; Put("(" &
integer'image((Navire.Equipage.max - Navire.Equipage.current) *
Reparateur_cost) &")") ;

  IF Indice = 3 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  Goto_Xy(X+36,Y-1) ; Put("ACHETER ") ;

  IF Indice = 5 THEN Set_Foreground(Light_Red) ; ELSE
  Set_Foreground(Blue) ; END IF ;
  IF Navire IN T_Tradeship'Class
    THEN Goto_Xy(X+48,Y-1) ; Put(" VENDRE") ;
    Goto_Xy(X+50,Y) ; Put("(" &
integer'image(T_Tradeship(Navire).stock * Goods_cost) &")") ;
  END IF ;

  Goto_Xy(0,Y+1) ;
  put(Blank_line) ;
  Goto_Xy(X+4+12*(Indice-1),Y+1) ;
  Set_Foreground(Light_Red) ;
  put(character'val(30)) ;
  Set_Foreground(Black) ;
END Put_Market_Menu1 ;

PROCEDURE Put_Market_Menu2(Indice : Natural := 1) IS
  X : constant Natural := 35 ;
  Y : constant Natural := 12 ;
  blank_line : constant string(1..80) := (others => ' ') ;
BEGIN
  FOR I IN 1..7 LOOP
    IF Indice = I THEN Set_Foreground(Light_Red) ; ELSE
    Set_Foreground(Blue) ; END IF ;
    Goto_Xy(0,Y-1) ;
    Put(Blank_Line) ;
    Goto_Xy(X,Y-1) ;
    CASE I IS
      WHEN 1 => Put("Gouvernail (" &
integer'image(Vitesse_cost) & " + 1 Vitesse)") ;
      WHEN 2 => Put(" B o u c l i e r s (" &
integer'image(Defense_cost) & " /marin + 1 Defense") ;
      WHEN 3 => Put(" B l i n d a g e (" &
integer'image(Cuirasse_cost) & " ) + 1 Cuirasse") ;
      WHEN 4 => Put("Sabres (" & integer'image(Attaque_cost)
& "/marin) + 1 Attaque") ;
      WHEN 5 => Put(" C a n n o n s (" &
integer'image(Puissance_cost) & " ) + 1 Puissance") ;
      WHEN 6 => Put("Couchettes (" &
integer'image(Equipage_cost) & " ) + 2 Equipage") ;
      WHEN 7 => Put("Planches (" & integer'image(Coque_cost)
& ") + 2 Coque") ;
    END CASE ;
  END LOOP ;

  Goto_Xy(X-2,Y-Indice) ;
  Set_Foreground(Light_Red) ;
  put(character'val(26)) ;
  Set_Foreground(Black) ;
END Put_Market_Menu2 ;

PROCEDURE Put_Select_Menu1(Indice : Natural := 1) IS
BEGIN
  FOR I IN 1..4 LOOP
    IF Indice = I
      THEN set_foreground(Light_Red) ;
      Goto_Xy(0,I-3) ;
      put(character'val(26)) ;
    ELSE set_foreground(Blue) ;
    END IF ;
    goto_xy(2,I+3) ;
    CASE I IS
      WHEN 1 => put("Navire de guerre") ;
      WHEN 2 => put("Navire marchand") ;
      WHEN 3 => put("Pirate") ;
      WHEN 4 => Put("Corsaire") ;
    END CASE ;
  END LOOP ;
  set_foreground(black) ;
END Put_Select_Menu1 ;

PROCEDURE Put_Select_Menu2(Indice : Natural := 1) IS
  color : color_type ;
BEGIN
  FOR I IN Liste_Navire'range LOOP
    IF Indice = I
      THEN color := Light_Red ;
      print(" " & character'val(26),color,20,I-3) ;
    ELSE Color := Blue ;
      print(" ",color,20,I-3) ;
    END IF ;
    print(to_string(Liste_Navire(i).nom),color,22,i+3) ;
  END LOOP ;
  set_foreground(black) ;
END Put_Select_Menu2 ;

```

P_Modes

Et enfin, le cœur du jeu : P_Mode. C'est dans ce dernier package que sont gérés les touches du clavier et la réaction à adopter.

Code : Ada - P_Modes.ads

```
WITH P_Navire ;           USE P_Navire ;
```

```

PACKAGE P_Modes IS
    FUNCTION First_Mode RETURN T_Navire'Class ;
    PROCEDURE Fight_Mode (Joueur : IN OUT T_Navire'Class) ;
    FUNCTION NextFight_Mode RETURN natural ;
    PROCEDURE Market_Mode(Joueur : IN OUT T_Navire'Class) ;
    PROCEDURE Buy_Mode(Joueur : IN OUT T_Navire'Class) ;
    FUNCTION Select_Mode RETURN T_Navire'Class ;
END P_Modes ;

```

Le nombre de procédures et fonctions est restreint : First_Mode() gère le tout premier menu pour choisir entre une nouvelle ou une ancienne partie ; en cas de nouvelle partie, il lancera Select_Mode() qui permet au joueur de choisir son navire et sa classe (marchand, pirate...); puis il accède au menu du marché, Market_Mode(), qui est le principal menu (c'est de la que se font les sauvegardes en appuyant sur Escap, les achats avec Buy_Mode(), les ventes, les répartitions ou le départ en mer avec Fight_Mode()); enfin, NextFight_Mode() gère un menu proposant au joueur de rentrer au port ou de continuer son aventure à la fin de chaque combat.

Secret (cliquez pour afficher)

Code : Ada - P_Modes.adb

```

WITH Ada.Text_IO ;      USE Ada.Text_IO ;
WITH Ada.Strings.Unbounded ;   USE Ada.Strings.Unbounded ;
WITH Nt_Console ;        USE Nt_Console ;
WITH P_Screen ;          USE P_Screen ;
WITH P_Variables ;       USE P_Variables ;
WITH P_Navire.List ;     USE P_Navire.List ;
WITH P_Data ;            USE P_Data ;

PACKAGE BODY P_Modes IS
    FUNCTION First_Mode RETURN T_Navire'Class IS
        Position : Natural := 1 ;
        touche : character ;
    BEGIN
        Main_Loop : LOOP
            Clear_Screen(White) ;
            Put_Title ;
            Put_First_Menu(Position) ;
            Touche := Get_Key ;

            IF Character'Pos(Touche) = 75 AND Position = 2      -- TOUCHE GAUCHE
            THEN Position := 1 ;
            ELSIF Character'Pos(Touche) = 77 AND Position = 1      -- TOUCHE DROITE
            THEN Position := 2 ;
            END IF ;

            IF Character'Pos(Touche) = 13
            THEN Print("Entrez le nom du fichier :",Blue,0,6) ;
                Save_File_Name := To_Unbounded_String(Get_Line) ;
                IF Position = 1
                THEN RETURN Select_Mode ;
                ELSIF Position = 2
                THEN RETURN
            Load_Navire(To_string(save_file_name)) ;
                END IF ;
            END IF ;

            END LOOP Main_Loop ;
        END First_Mode ;

        PROCEDURE Fight_Mode(Joueur : IN OUT T_Navire'Class) IS
            Touche : Character ;
            Position : Natural := 4 ;
            N : T_Percentage ;
            Ordi : T_Navire'Class := generer_ennemi(Niveau) ;
        BEGIN
            Main_loop : LOOP
                Raz_bonus(Joueur) ;
                Clear_Screen(White) ;
                Put_Title ;
                Put_Status(Joueur,Gauche) ;
                Put_Status(Ordi,Droite) ;
                IF Joueur.Est_Mort
                THEN Joueur.Perd ;
                EXIT ;
                END IF ;

                ----- TOUR JOUEUR HUMAIN -----
                LOOP
                    Put_Fight_Menu(Joueur,Position) ;
                    Touche := Get_Key ;
                    --touche gauche
                    IF Character'Pos(Touche) = 75
                    THEN IF Position = 1
                        OR (Position = 2 AND Joueur NOT IN
                            T_Illegal'Class)
                        THEN NULL ;
                        ELSIF Position = 3 AND Joueur NOT IN
                            T_Legal'Class
                        THEN Position := Position - 2 ;
                        ELSE Position := Position - 1 ;
                        END IF ;
                    END IF ;
                    --touche droite
                    IF Character'Pos(Touche) = 77 and position < 5
                    THEN IF Position = 1 AND Joueur NOT IN
                            T_Legal'Class
                        THEN Position := Position + 2 ;
                        ELSE Position := Position + 1 ;
                        END IF ;
                    END IF ;
                    --touche entrée
                    IF Character'Pos(Touche) = 13
                    THEN CASE Position IS
                            WHEN 1 => Joueur.Aborde(Ordi) ;
                            WHEN 2 => joueur.bombarde(ordi) ;
                            WHEN 3 => joueur.defend ;
                            WHEN 4 => joueur.maneuvre ;
                            WHEN OTHERS => put_fuite_message ; exit
                        Main_loop ;
                        END CASE ;
                        EXIT ;
                    END IF ;
                END LOOP ;

                ----- TOUR ORDINATEUR --
                Raz_bonus(Ordi) ;
                Clear_Screen(White) ;
                Put_Title ;
                Put_Status(Joueur,Gauche) ;
                Put_Status(Ordi,Droite) ;
                IF Ordi.Est_Mort
                THEN Joueur.Bat(Ordi) ;
                EXIT ;
                END IF ;

                N := Random ;
                IF N >= 90
                THEN Ordi.Maneuvre ;
                ELSIF N>=80
                THEN Ordi.Defend ;
                ELSIF Ordi IN T_Legal'Class AND Ordi IN T_Illegal'Class
                THEN IF N mod 2 = 0
                    THEN Ordi.Bombarde(Joueur) ;
                    ELSE Ordi.Aborde(Joueur) ;
                    END IF ;
                ELSIF Ordi IN T_Legal'Class
                THEN Ordi.Bombarde(Joueur) ;
                else ordi.aborde(joueur) ;
                END IF ;
            END IF ;
        END Fight_Mode ;
    
```

```

        END IF ;
    END LOOP Main_loop ;
END Fight_Mode ;

FUNCTION NextFight_Mode RETURN natural IS
    Position : Natural := 1 ;
    touche : character ;
BEGIN
    LOOP
        Put_NextFight_Menu(position) ;
        Touche := Get_Key ;
        --TOUCHE GAUCHE
        IF Character'Pos(Touche) = 75 AND Position > 1
            THEN Position := Position - 1 ;
        END IF ;
        --TOUCHE DROITE
        IF Character'Pos(Touche) = 77 AND Position < 3
            THEN Position := Position + 1 ;
        END IF ;
        IF Character'Pos(Touche) = 13 OR Character'Pos(Touche) =
72
            THEN CASE Position IS
                WHEN 1 => Niveau := 1 ;
                WHEN 3 => Niveau := Niveau + 1 ;
                WHEN OTHERS => null ;
            END CASE ;
            RETURN Position ;
        END IF ;
    END LOOP ;
END NextFight_Mode ;

PROCEDURE Market_Mode(Joueur : IN OUT T_Navire'Class) IS
    Position : Natural := 1 ;
    Touche : character ;
BEGIN
    Main_Loop : LOOP
        Raz_bonus(Joueur) ;
        Clear_Screen(White) ;
        Put_Title ;
        Put_Status(Joueur,Gauche) ;
        Put_Gold(joueur) ;

        LOOP
            Put_Market_Menu1(Joueur,Position) ;
            Touche := Get_Key ;
            --touche gauche
            IF Character'Pos(Touche) = 75 and position > 1
                THEN Position := Position - 1 ;
            END IF ;
            --touche droite
            IF Character'Pos(Touche) = 77 and ((Joueur in
T_Tradeship'class and position = 4) or position < 4)
                THEN Position := Position + 1 ;
            END IF ;
            --touche echap
            IF Character'Pos(Touche) = 27
                THEN Save_Navire(Joueur) ;
                Print("Fichier sauvegardé sous " &
To_String(Save_File_Name) & ".txt",Red,0,24) ;
                sleep ;
                EXIT Main_Loop ;
            END IF ;
            --touche entrée
            IF Character'Pos(Touche) = 13 or Character'Pos(Touche)
= 72
                THEN CASE Position IS
                    WHEN 1 => Fight_Loop : loop
                        Fight_Mode(Joueur) ;
                        IF Joueur.Est_Mort
                            THEN touche := get_key ;
                            EXIT Main_Loop ;
                        ELSE EXIT Fight_Loop ;
                    END IF ;
                    WHEN 2 => Joueur.reparer ;
                    WHEN 3 => Joueur.recruter ;
                    WHEN 4 => Buy_Mode(joueur) ;
                    WHEN OTHERS => null ;
                END CASE ;
                T_Tradeship(Joueur).vendre ;
                END IF ;
            END IF ;
        END LOOP Main_Loop ;
    END Market_Mode ;

PROCEDURE Buy_Mode(Joueur : IN OUT T_Navire'Class) IS
    Position : Natural := 1 ;
    Touche : character ;
BEGIN
    while position > 0 LOOP
        Put_Market_Menu2(Position) ;
        Touche := Get_Key ;
        --touche haut
        IF Character'Pos(Touche) = 72 and position < 7
            THEN Position := Position + 1 ;
        END IF ;
        --touche bas
        IF Character'Pos(Touche) = 80 and position > 0
            THEN Position := Position - 1 ;
        END IF ;
        --touche gauche ou droite
        IF Character'Pos(Touche) = 75 or Character'Pos(Touche) =
77
            THEN Position := 0 ;
        END IF ;
        --touche entrée
        IF Character'Pos(Touche) = 13
            THEN CASE Position IS
                WHEN 1 => Joueur.Ameliorer_vitesse ;
                WHEN 2 => Joueur.Ameliorer_defense ;
                WHEN 3 => Joueur.Ameliorer_cuirasse ;
                WHEN 4 => Joueur.Ameliorer_attaque ;
                WHEN 5 => Joueur.Ameliorer_puissance ;
                WHEN 6 => Joueur.Ameliorer_equipage ;
                when 7 => Joueur.Ameliorer_coque ;
                WHEN OTHERS => null ;
            END CASE ;
            Put_Gold(joueur) ;
            Put_Status(Joueur,Gauche) ;
        END IF ;
    END LOOP ;
END Buy_Mode ;

FUNCTION Select_Mode RETURN T_Navire'Class IS
    Navire : ACCESS T_Navire'Class ;
    Pos1,Pos2 : Natural := 1 ;
    touche : character ;
BEGIN
    Main_Loop : LOOP
        Menu1 : LOOP
            Clear_Screen(White) ;
            Put_Title ;
            Put_Select_Menu1(Pos1) ;
            Touche := Get_Key ;
            --touche haut
            IF Character'Pos(Touche) = 72 and pos1 > 1
                THEN pos1 := pos1 - 1 ;
            END IF ;
            --touche bas
            IF Character'Pos(Touche) = 80 and pos1 < 4
                THEN pos1 := pos1 + 1 ;
            END IF ;
            --touche entrée ou droite
            IF Character'Pos(Touche) = 13 or Character'Pos(Touche)
= 77
                THEN sleep ; EXIT Menu1 ;
            END IF ;
        END LOOP Menu1 ;
    END LOOP Main_Loop ;
END Select_Mode ;

```

```

Menu2 : LOOP
  Put_Select_Menu2(Pos2) ;
  Touche := Get_Key ;
  --touche haut
  IF Character'Pos(Touche) = 72 AND pos2 > 1
    THEN pos2 := pos2 - 1 ;
  END IF ;
  --touche bas
  IF Character'Pos(Touche) = 80 AND pos2 <
Liste_Navire'length
    THEN pos2 := pos2 + 1 ;
  END IF ;
  --touche entrée ou droite
  IF Character'Pos(Touche) = 13 OR Character'Pos(Touche)
= 77
    THEN beep ; EXIT Main_Loop ;
  END IF;
  IF Character'Pos(Touche) = 75
    THEN EXIT Menu2 ;
  END IF;
  END LOOP Menu2 ;
END LOOP Main_Loop ;

CASE Pos1 IS
  WHEN 1 => Navire := NEW
    T_Warship'(Init(Liste_Navire(Pos2).nom, Liste_Navire(Pos2).stat))
  ;
  WHEN 2 => Navire := NEW
    T_Tradeship'(Init(Liste_Navire(Pos2).nom,
Liste_Navire(Pos2).stat)) ;
  WHEN 3 => Navire := NEW
    T_Pirate'(Init(Liste_Navire(Pos2).nom, Liste_Navire(Pos2).stat)) ;
  WHEN others => Navire := NEW
    T_Corsair'(Init(Liste_Navire(Pos2).nom, Liste_Navire(Pos2).stat))
  ;
END CASE ;
  Navire.Playable := True ;
  Navire.Gold := 100 ;
  return Navire.all ;
END Select_Mode ;

```

Pistes d'amélioration :

- Ajouter un écran affichant les meilleurs scores obtenus, les navires les plus puissants, les plus riches, les plus peuplés...
- Ajouter une nationalité à votre navire (français, anglais, portugais, hollandais, espagnol, pirate...) ajoutant de nouveaux bonus/malus ou proposant des navires spécifiques.
- Vous pouvez également approfondir le système de jeu en proposant de personnaliser les armes (grappins, filets, sabres, lances, petits ou gros canons...) ou les manœuvres, ou encore de changer pour un navire plus performant.
- Pourquoi ne pas proposer un jeu en deux temps : des phases de bataille navale (ce que vous venez de créer) et des phases d'exploration où le joueur devrait déplacer son navire sur une carte.

Les exceptions

Je me suis efforcé depuis les premiers chapitres de vous apprendre à rédiger correctement vos algorithmes afin d'éviter toute erreur. Je vous ai régulièrement incité à penser aux cas particuliers, aux bornes des intervalles des tableaux, aux conditions de terminaison des boucles itératives ou récursives... Malheureusement, il y a des choses que vous ne pouvez pas prévoir car elles dépendent du programmeur mais de l'utilisateur : un fichier nécessaire à votre logiciel qui aurait été malencontreusement supprimé, un utilisateur qui n'aurait pas compris que le nombre entier demandé par le programme ne peut pas avoir de virgule ou que 'A' n'est pas un nombre, la mémoire est saturée...

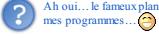
Vos codes, pour l'heure, peuvent gérer certains de ces problèmes mais cela vous oblige généralement à les bidouiller et à les rendre illisibles (usage de boucles, saisie de texte converti ensuite en nombre...) Et qui plus est, vous ne pouvez gérer tous les cas de plantage. Heureusement, le langage Ada peut gérer ces erreurs grâce à ce que l'on appelle les **exceptions**. Les exceptions permettent de signaler le type d'erreur commis voire même de les traiter et de rétablir le fonctionnement normal du programme. Le langage Ada est même réputé pour la gestion des erreurs.

Fonctionnement d'une exception

Vous avez dit exception ?

Vous avez du régulièrement vu retrouver face au message suivant : `raised CONSTRAINT_ERROR : ###.abd:...`

Je vous ai même parfois incité à écrire des programmes menant à ce genre d'erreur.



Ah oui... le fameux plantage du Constraint_error ! Je ne sais même plus combien de fois il a accompagné le crash de mes programmes... 😊

C'est certainement l'une des exceptions les plus courantes au début. L'exception `CONSTRAINT_ERROR`, indique que vous tentez d'accéder à un élément inexistant (cible d'un pointeur nul, accès à la 11ème valeur d'un tableau à 10 éléments...) ou qu'un calcul risque de générer un overflow (pour des rappels sur l'overflow, voir le premier exercice du chapitre sur la récursivité ou bien le chapitre traitant de la représentation des nombres entiers). Mais vous ne devez pas la voir comme un plantage de votre programme. En fait, Ada se rend compte à un moment donné qu'il y a une erreur et lève une exception afin d'éviter un véritable crash, c'est à dire le programme qui se ferme tout seul sans vous dire merci et votre système d'exploitation qui vous indique qu'il a rencontré un petit soucis.



Et ça nous avance à quoi que Ada s'en rende compte puisque de toutes façons le résultat est le même ? 😊

Que vous êtes défaitistes, c'est au contraire d'une grande aide :

- Ada nous indique de quel type d'erreur il s'agit tout d'abord. Il nous indique également la ligne du code où le souci est apparu et ajoute quelques informations supplémentaires. Ces indications sont très utiles à la fois pour la personne qui développe le programme, mais aussi pour celle qui sera chargée de sa maintenance.
- Une fois l'exception connue, il sera possible d'effectuer un traitement particulier adapté au type d'erreur rencontré, et parfois même de résoudre ce problème afin de rétablir un fonctionnement normal du programme.

Une exception n'est donc pas un simple plantage de votre programme mais plutôt une alerte concernant un événement exceptionnel nécessitant un traitement approprié. Jusque là, aucun traitement n'étant proposé, Ada décida de mettre lui-même fin au programme pour éviter tout problème.

Le fonctionnement par l'exemple

Testons par exemple le code suivant afin de comprendre ce qu'il se passe :

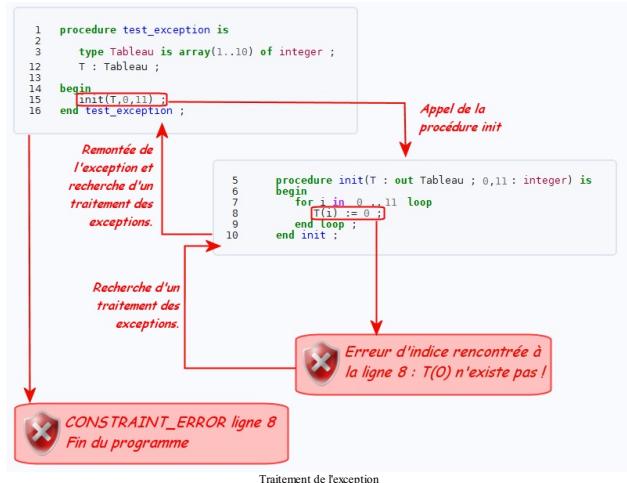
```
Code : Ada
procedure test_exception is
    type Tableau is array(1..10) of integer;
    procedure init(T : out Tableau; dbt, fin : integer) is
    begin
        for i in dbt..fin loop
            T(i) := 0;
        end loop;
    end init;
    T : Tableau;
begin
    init(T,0,11);      --Il y aura un gros soucis à ce moment là !
end test_exception;
```

Mais devriez obtenir le doux message suivant :
`raised CONSTRAINT_ERROR : test_exception.adb:8 index check failed.` Traduction : Ada a levé (raised) une exception de type `CONSTRAINT_ERROR`. Celle-ci a été déclenchée lors de l'exécution de l'instruction située à la ligne 8 de `test_exception.adb`. Information complémentaire : vous avez commis une erreur sur les indices du tableau.



Sauf que l'erreur dans ce code est à la ligne 15 et pas 8 ! Quelle idée d'initialiser 12 éléments dans un tableau de 10 ?!

Reprendons le cheminement tranquillement. Votre programme commence par effectuer ses déclarations de types, variables et procédures. Jusque là, rien de catastrophique. Puis il exécute l'instruction `Init()` durant laquelle une boucle est exécutée qui demande l'accès en écriture à l'élément n°0 de notre tableau. C'est cette demande qui lève une exception. Que fait le programme ? Il cherche, au sein de la procédure `Init()`, une proposition de traitement lié à ce type d'erreur. Puisqu'il n'en trouve pas, il met fin à la procédure et répercute l'exception dans la procédure principale et recommence. Il cherche à nouveau un traitement lié à cette exception et comme il n'y en a toujours pas, il met fin à cette nouvelle procédure. Et comme il s'agit de la procédure principale, le programme prend fin en affichant l'exception rencontrée.



Comme le montre le schéma ci-dessous, la levée d'une exception ne génère pas automatiquement l'arrêt brutal de votre programme, mais plutôt un arrêt en chaîne si aucun traitement approprié n'a été prévu. Votre programme cherchera un remède à l'exception rencontrée au sein même du sous-programme responsable avant de transmettre ses ennuis au sous-programme qui l'avait auparavant appelé.

Traitement d'une exception

Le bloc EXCEPTION

Vous aurez également remarqué sur le schéma que les flèches indiquant la recherche d'un traitement des exceptions pointent

toutes vers la fin des procédures. En effet, la levée d'une exception dans une procédure ou une fonction entraîne immédiatement son arrêt. Le programme «saute» alors à la fin de la procédure ou fonction pour y chercher un éventuel traitement. C'est donc à la toute fin de nos sous-programmes que nous allons tâcher de régler nos erreurs à l'aide du mot clé **EXCEPTION**. Vous allez voir, ce nouveau mot-clé a un fonctionnement analogue à celui de **CASE**:

Code : Ada

```
procedure init(T : out Tableau ; dbt, fin : integer) is
begin
  for i in dbt..fin loop
    T(i) := 0;
  end loop;
exception
  when CONSTRAINT_ERROR => Put_line("Tu pouvais pas réfléchir avant
d'écrire tes indices ?");
  when others => Put_line("Qu'est-ce que t'as encore fait comme anerie
?");
end init;
```

Nous nous contenterons pour l'instant d'un simple affichage. Si vous testez ce nouveau code, vous devriez obtenir le complément suivant : **Tu pouvais pas réfléchir avant d'écrire tes indices ?**. Cela ne règle toujours pas notre problème mais nous avons progressé : plus de vilains affichages agressifs et incompréhensibles. Une autre solution aurait consisté à gérer l'exception non pas dans la procédure **init()** mais dans la procédure principale :

Code : Ada

```
procedure test_exception is
  type Tableau is array(1..10) of integer;
  procedure init(T : out Tableau ; dbt, fin : integer) is
  begin
    for i in dbt..fin loop
      T(i) := 0;
    end loop;
  end init;
  T : Tableau;
begin
  init(T,0,11);
exception
  when CONSTRAINT_ERROR => Put_line("Tu pouvais pas réfléchir avant
d'écrire tes indices ?");
  when others => Put_line("Qu'est-ce que t'as encore fait
comme anerie ?");
end test_exception;
```

Où et quand gérer une exception

Mouais, bah ça c'est typiquement le genre d'exemple inutile que j'aurais pu trouver tout seul. Tu chercheras pas à meubler parce que tu n'as plus rien à dire 😊

Vous croyez ? Eh bien non, je n'essaie pas de meubler mais bien de vous présenter deux cas différents. Alors oui, je sais : au final, le programme affiche toujours la même phrase. Alors pour comprendre la différence, ajoutez cette ligne juste après « **init(T,0,11)** » et avant la fin du programme et le bloc de gestion des exceptions :

Code : Ada

```
put_line("Initialisation réussie !");
```

Testez ensuite les deux cas. Quand le bloc **EXCEPTION** se trouve dans la procédure principale, rien ne change ! La procédure **init()** échoue toujours entraînant l'affichage de cette fameuse phrase de congratulation. Mais lorsque le bloc **EXCEPTION** se trouve au sein même de la procédure **init()**, l'affichage obtenu est tout autre :

Code : Console

```
Tu pouvais pas réfléchir avant d'écrire tes indices ?
Initialisation réussie !
```

Eh ouï ! Le programme principal reprend son cours, comme si de rien n'était. Victoire ! Il ne plante plus !

Wouhouh ! Mais au fait, que vaut le tableau alors ?

C'est justement sur ce point que je souhaitais attirer votre attention. La levée de l'exception n'entraîne plus l'arrêt du programme grâce au traitement effectué, mais le problème n'a pas été résolu. Le programme principal continue donc comme si de rien n'était alors que le tableau qu'il était sensé initialiser n'a jamais vu la moindre affectation. Tentez un tri sur ce tableau ou la moindre lecture, et vous aurez droit de nouveau à une jolie exception. De ce simple exemple, nous devons tirer quelques enseignements :

- Soit nous traitons cette erreur dans la procédure en cause, **init()**, et alors nous traitons le problème à fond afin de retourner un tableau vraiment initialisé :

Code : Ada

```
EXCEPTION
  WHEN CONSTRAINT_ERROR => Put_line("Il y a eu un petit
contre-temps, mais c'est fait.");
  T := (OTHERS => 0);
```

- Soit nous propagons l'exception au programme principal sciemment à l'aide du mot clé **RAISE** :

Code : Ada

```
EXCEPTION
  WHEN CONSTRAINT_ERROR => Put_line("Il y a eu un petit
soucis.");
  RAISE;
```

- Soit nous ne nous en préoccupons que dans le programme principal en considérant que le problème provient de là et qu'il doit donc être traité par la procédure **test_exception()**.

Enfin, quel que soit le cas de figure que vous choisirez, gardez en mémoire le petit schéma vu précédemment : avant de traiter une exception vous devez réfléchir à sa propagation et au meilleur moment de la traiter.

Exceptions prédefinies Exceptions standards

Le langage Ada dispose nativement de nombreuses exceptions prédefinies. Nous avons jusque là traité de l'exception **CONSTRAINT_ERROR** qui est levée lors d'une erreur sur l'indexation d'un tableau, lors de l'accès à un pointeur **NULL**, d'un overflow, d'une opération impossible (division par 0 par exemple). Mais d'autres existent et sont définies par le package Standard :

- **PROGRAM_ERROR** : celle-là, vous ne l'avez probablement jamais vue. Elle est appellée lorsqu'un programme fait appel à un sous-programme dont le corps n'a pas encore été rédigé ou lorsqu'une fonction se termine sans être passée par la case **RETURN**. Cela peut arriver notamment lors d'une erreur avec une fonction récursive (nous verrons un exemple juste après).
- **STORAGE_ERROR** : cette exception est levée lorsqu'un programme exige plus de mémoire qu'il n'en a le droit. Cela arrive régulièrement avec des programmes récursifs mal ficelés : une boucle récursive infinie est engendrée, nécessitant plus de mémoire à chaque appel. L'ordinateur met fin à cette gabegie lorsque la mémoire demandée devient trop importante.
- **TASKING_ERROR** : Celle-là non plus, vous n'avez pas du la voir beaucoup, et pour cause elle est levée lorsqu'un problème de communication apparaît entre deux tâches. Mais comme vous ne savez pas encore ce qu'est une tâche en

Ada, cela ne vous avance pas à grand chose, patience.

Exceptions et fonctions récursives

Avant de présenter de nouvelles exceptions, attendons-nous sur le cas des fonctions récursives évoqué ci-dessus. Voici une fonction toute simple qui divise un nombre par 4, 3, 2, 1 puis 0. Bien entendu, la division par 0 engendrera une exception :

Code : Ada

```
with ada.integer_text_io ;           use ada.integer_Text_IO ;
procedure Test_exception_recursive is
    function division(a : integer ; n : integer) return integer is
        begin
            put(a) ;
            return division(a/n,n-1) ;
        end division ;
begin
    put(division(125, 4) ) ;
end test_exception_recursive ;
```

Comme attendu, nous nous retrouvons avec une belle exception :

Code : Console

```
125      31      10      5      5
raised CONSTRAINT_ERROR : test_exception_recursive.adb:8 divide by zero
```

Nous allons résoudre ce problème à l'aide de nos exceptions (oui, je sais, il y a plus simple mais c'est un exemple) :

Code : Ada

```
with ada.Text_IO ; use ada.Text_IO ;
with ada.integer_text_io ;           use ada.integer_Text_IO ;
procedure Test_exception_recursive is
    function division(a : integer ; n : integer) return integer is
        begin
            put(a) ;
            return division(a/n,n-1) ;
        exception
            when others => Put_Line("Division par zero !") ;
        end division ;
begin
    put(division(125, 4) ) ;
end test_exception_recursive ;
```

Et que se passe-t-il désormais ?

Code : Console

```
125      31      10      5      5Division par zero !
Division par zero !
Division par zero !
Division par zero !
Division par zero !
raised PROGRAM_ERROR : test_exception_recursive.adb:8 missing return
```



Pourquoi Ada lève-t-il une exception ? On les avait toutes traitées avec EXCEPTION et WHEN OTHERS, non ?

Pas tout à fait. Nous avons réglé le cas des exceptions levées lors de la fonction. Notre fonction division() n°1 (avec n valant 4) effectue quatre appels récursifs. Et l'appel division() n°5 (avec n valant 0) lève une exception (la division par zéro) au lieu de renvoyer un integer. Du coup, le même phénomène se produit pour les division() n°4, 3, 2 puis 1 ! Tout cela est très bien, mais il n'empêche que notre programme principal attend que la fonction lui renvoie un résultat, ce qu'elle ne peut pas faire puisque notre traitement des exceptions ne contient pas d'instruction RETURN. Ce qui lève donc cette fameuse exception : PROGRAM_ERROR. Encore une fois, il est important de réfléchir à la propagation d'une exception avant de se lancer dans un traitement hasardeux.

Autres exceptions prédéfinies

D'autres exceptions sont définies par le langage dans divers packages. Mais vous avez du en rencontrer beaucoup lorsque vous avez travaillé sur les fichiers : lorsque le mode n'était pas bon, que vous n'aviez pas pensé à gérer les fins de fichiers, ou que le programme ne trouvait pas votre fichier pour une toute petite erreur d'*hautogaffe*. Ces exceptions concernant les entrées-sorties sont définies dans le package Ada.IO_Exceptions (a-iocxe.ads) :

- **Status_Error** : levée lorsque le status d'un fichier (ouvert/fermé) est erroné. Par exemple, fermer ou lire un fichier déjà fermé.
- **Mode_Error** : levée lorsque le mode d'un fichier (in_file/out_file/append_file) est erroné. Par exemple, si vous tentez d'écrire dans un fichier en lecture seule.
- **Name_Error** : levée lorsque le nom d'un fichier est erroné, c'est-à-dire quand il est introuvable.
- **Use_Error** : levée lorsque l'usage d'un fichier ne correspond pas à son type. Pour rappel, nous avions vu trois types de fichiers : texte, séquentiel et à accès direct.
- **Device_Error** : levée lors de problèmes liés au matériel.
- **End_Error** : levée lorsque vous tentez de lire au-delà de la fin du fichier (pensez à utiliser End_of_page(), End_of_line() et End_of_file()).
- **Data_Error** : levée lorsque l'utilisateur fournit des données du mauvais type. Par exemple, lorsque le programme demande un entier et que l'utilisateur lui renvoie un '*A*'.
- **Layout_Error** : levée lorsqu'un string est trop long pour être affiché.

Créer et lever ses propres exceptions

Déclarer une exception

Supposons que vous soyez en pleine conception d'un programme d'authentification. Votre programme doit saisir un identifiant composé de 5 lettres ainsi qu'un mot de passe comportant 6 lettres suivies de 2 chiffres et les enregistrer dans un fichier. La partie sensible se pose au moment de la saisie du mot de passe : on ne va pas enregistrer un mot de passe incorrect ! Vous avez donc rédigé pour cela une fonction mdp_correct() qui renvoie FALSE en cas d'erreur.

Et justement s'il y a erreur, votre programme devra lever une exception. Nous appellerons notre exception PWD_ERROR. Notez au passage que j'écrirai toujours mes exceptions en majuscules et les nommerai toujours avec le suffixe _ERROR. Une autre solution consiste à les préfixer avec EXC_ de la même manière que nos types étaient préfixés par un T_ et nos packages par un P_.

La déclaration de l'exception se fait en même temps que les déclarations de types ou de variables de la façon suivante :

Code : Ada

```
PWD_ERROR : EXCEPTION ;
```

Lever sa propre exception



C'est bien gentil d'avoir ses propres exceptions, mais comment Ada peut-il savoir quand la lever ? Ce n'est pas une exception standard !

Cela se fera à l'aide du mot clé **RAISE**, déjà vu précédemment. Voici donc un exemple de programme :

Code : Ada

```
procedure Authentification is
  PWD_ERROR : exception ;
  login      : string(1..5) ;
  pwd        : string(1..8) ;
begin
  login := saisie_login ;
  pwd   := saisie_pwd ;
  if mdp_correct(pwd)
    then Enregistrement(login, pwd) ;
  else raise PWD_ERROR ;
  end if ;
exception
when PWD_ERROR => put_line("Mot de passe incorrect !") ;
when others => put_line("Le programme a rencontré une erreur fatale !
") ;
end Authentification ;
```

Et de la même manière qu'avec une exception standard, l'instruction **RAISE** va mettre fin à notre procédure et effectuer un bond vers le bloc de traitement pour trouver une réponse adaptée. Il est également possible avec l'instruction **WITH** d'indiquer immédiatement à la levée de l'exception le message d'alerte qui sera affiché.

Code : Ada

```
procedure Authentification is
  PWD_ERROR : exception ;
  login      : string(1..5) ;
  pwd        : string(1..8) ;
begin
  login := saisie_login ;
  pwd   := saisie_pwd ;
  if mdp_correct(pwd)
    then Enregistrement(login, pwd) ;
  else raise PWD_ERROR with "Mot de passe incorrect !" ;
  end if ;
exception
when PWD_ERROR => null ;
when others => put_line("Le programme a rencontré une erreur fatale !
") ;
end Authentification ;
```

Propagation de l'exception

 Mais si la procédure **Authentification()** fait partie d'un programme plus vaste, que va-t-il se passer pour le programme principal ? L'authentification n'aura pas eu lieu et le programme principal ne connaîtra pas mon exception puisqu'elle est définie seulement dans la sous-procédure.

Une solution consiste à propager l'exception aux programmes appelants en écrivant :

Code : Ada

```
WHEN PWD_ERROR => put_line("Mot de passe incorrect !") ; RAISE ;
```

Mais le programme principal ne connaîtra pas **PWD_ERROR**, car sa portée est limitée à la procédure **Authentification()**. Le programme principal devra donc comporter un bloc **EXCEPTION** dans lequel **PWD_ERROR** sera récupéré par l'instruction **WHEN OTHERS**.

Rectifier son code

Le retour du bloc **DECLARE**

 Tout cela c'est bien beau, mais ça ne répare pas les bêtises. Il vaudrait mieux demander à l'utilisateur de recommencer plutôt que de propager **PWD_ERROR**. Mais comment faire puisque le bloc **EXCEPTION** est forcément à la fin de la procédure ?

Attention, le bloc **EXCEPTION** doit se trouver à la fin d'un autre bloc mais pas nécessairement à la fin du sous-programme. Il peut donc terminer une **FUNCTION**, une **PROCEDURE** ou un **DECLARE**. Et c'est ce dernier que nous allons utiliser :

Code : Ada

```
procedure Authentification is
  login      : string(1..5) ;
  pwd        : string(1..8) ;
begin
  login := saisie_login ;
  pwd   := saisie_pwd ;
  declare
    PWD_ERROR : exception ;
  begin
    if mdp_correct(pwd)
      then Enregistrement(login, pwd) ;
    else raise PWD_ERROR ;
    end if ;
  exception
  when PWD_ERROR => put_line("Mot de passe incorrect !") ;
  when others => put_line("Le programme a rencontré une erreur fatale !
") ;
  end ;
end Authentification ;
```

Une autre façon, plus élégante et moins lourde consiste à créer un simple bloc introduit par **BEGIN** sans rien déclarer :

Code : Ada

```
procedure Authentification is
  PWD_ERROR : exception ;
  login      : string(1..5) ;
  pwd        : string(1..8) ;
begin
  login := saisie_login ;
  pwd   := saisie_pwd ;
  TRY begin
    if mdp_correct(pwd)
      then Enregistrement(login, pwd) ;
    else raise PWD_ERROR ;
    end if ;
  exception
  when PWD_ERROR => put_line("Mot de passe incorrect !") ;
  when others => put_line("Le programme a rencontré une erreur fatale !
") ;
  end TRY ;
end Authentification ;
```

Je conserverai cette seconde solution par la suite car elle a le mérite de ne rien déclarer en cours de route (pratique à bannir autant que possible) et de faire apparaître clairement un bloc de test (rappelons que « **try** » signifie « essayer ») dans lequel pourra être levé une exception. On parle alors de zone critique. Vénons-en maintenant à la façon de rectifier notre code. Il y a deux grandes options.

Les boucles

La première option consiste à enfermer notre section critique dans une boucle dont on ne sort qu'en cas de réussite :

Code : Ada

```

loop
  TRY : begin
    if mdp_correct(pwd)
      then Enregistrement(login, pwd) ;
        exit ; -- on peut même clarifier ce
    else raise PWD_ERROR ;
    end if ;
  exception
    when PWD_ERROR => put_line("Mot de passe incorrect !");
    when others => put_line("Le programme a rencontré une
erreur fatale !");
      raise ;
  end TRY ;
end loop ;

```

Si le mot de passe est valide, il est enregistré et le programme sort de la boucle. La levée de l'exception empêche donc la sortie de boucle et la portion de code est ainsi répétée. Une variante pourrait consister à proposer seulement trois essais et à clore le programme ensuite.

GOTO is back !

Cette solution, si elle a ses adeptes, me semble plutôt lourde et oblige à ouvrir deux blocs : un bloc itératif et un bloc déclaratif ! Et par conséquent, cela nous oblige à fermer deux blocs. Question efficacité, on a déjà fait mieux. 😊 Ma solution privilégiée est de faire appel à une instruction que je vous avais jadis interdite : **GOTO** ! 🎉

Son inconvénient majeur était qu'elle ne réalisait pas de blocs clairement identifiables et générât généralement du code dit «spaghetti» où les chemins suivis par le programme s'entremêlent au point que plus personne n'ose y mettre le nez. Mais dans le cas présent nous avons créé un bloc spécifique, alors profitons-en pour faire appel (proprement) à cette vieille instruction :

Code : Ada

```

<<TRY>>
begin
  if mdp_correct(pwd)
    then Enregistrement(login, pwd) ;
    else raise PWD_ERROR ;
  end if ;
exception
  when PWD_ERROR => put_line("Mot de passe incorrect !");
  when others => goto TRY ;
    raise ;
end ;

```

Assertions et contrats

Assertions



Après les exceptions, les assertions ! Qu'est-ce que c'est encore que ça ?

Une assertion est simplement un test rapide permettant de contrôler certains paramètres de votre programme et de lever une exception particulière (ASSERTION_ERROR) si les conditions ne sont pas remplies. Mais avant de vous expliquer sa mise en oeuvre, je dois vous parler des **PRAGMA**.

Peut-être avez-vous déjà vu cette instruction en feuilletant divers packages sans en comprendre le sens. Les **PRAGMA** sont ce que l'on appelle en français des directives de compilateur, c'est-à-dire des instructions qui s'adressent non pas au programme et à ses utilisateurs mais à votre bon vieux compilateur GNAT. Elle sont là pour lui donner des instructions sur la façon de compiler, de comprendre votre code, de le contrôler...

Mais ces directives sont également là pour éprouver votre code, vous apporter des garanties de fiabilité. Elles peuvent s'écrire durant les déclarationS ou le déroulement de vos programmes et s'utilisent de la manière suivante :

Code : Ada

```
PRAGMA Nom_de_directive(parametres_eventuels) ;
```

Il existe des dizaines et des dizaines de directives différentes. Certains compilateurs fournissent d'ailleurs leurs propres directives. Celle dont nous allons parler est heureusement utilisée par tous les compilateurs, il s'agit de la directive **Assert** (comme assertion, vous l'aurez compris). Reprenons le cas de notre mot de passe. Plutôt que de lever notre propre exception, il est possible de vérifier l'assertion (ou affirmation) suivante : «le mot de passe est correct». Si cette assertion est vérifiée, on continue notre bout de chemin. Sinon, l'exception ASSERTION_ERROR sera automatiquement levée. Reprenons notre exemple :

Code : Ada

```

procedure Authentification is
  login   : string(1..5) ;
  pwd    : string(1..8) ;
begin
  login := saisie_login ;
  pwd  := saisie_pwd ;
  pragma assert(mdp_correct(pwd)) ;
  Enregistrement(login, pwd) ;
exception
  when ASSERTION_ERROR => put_line("Mot de passe incorrect !");
  when others    => put_line("Le programme a rencontré une erreur
fatale !");
end Authentification ;

```

La directive Assert prend en paramètre une expression booléenne. Il est ainsi possible de vérifier la valeur d'un nombre ou d'effectuer un test d'appartenance en écrivant :

Code : Ada

```

PRAGMA Assert(x=0) ;
-- OU
PRAGMA Assert(MonObjet IN MonType'class) ;

```

Assert peut également prendre un second paramètre. On indique alors avec une chaîne de caractères le message à afficher en cas d'erreur :

Code : Ada

```

procedure Authentification is
  login   : string(1..5) ;
  pwd    : string(1..8) ;
begin
  login := saisie_login ;
  pwd  := saisie_pwd ;
  pragma assert(mdp_correct(pwd), "Mot de passe incorrect !");
  Enregistrement(login, pwd) ;
exception
  when ASSERTION_ERROR => null ;
  when others    => put_line("Le programme a rencontré une erreur
fatale !");
end Authentification ;

```

Supprimer les contrôles du compilateur

Tant que nous parlons de **PRAGMA**, je vais vous présenter une seconde directive de compilateur : suppress. Cette directive permet de lever le contrôle de certaines exceptions effectuée par le compilateur.

Par exemple :

Code : Ada

```
PRAGMA suppress(overflow_check) ;
```

Supprimez le test d'overflow lié à l'exception CONSTRAINT_ERROR sur le bloc entier. De même, si vous disposez d'un type T_Tableau, vous pourrez supprimer le contrôle des indices sur tous les types tableaux de tout le bloc en écrivant :

Code : Ada

```
PRAGMA suppress(index_check) ;
```

ou bien ne supprimer ces tests que sur les variables composites de types T_Tableau en écrivant :

Code : Ada

```
PRAGMA suppress(overflow_check, T_Tableau) ;
```

Voici les tests de l'exception CONSTRAINT_ERROR que vous pourrez supprimer avec « **PRAGMA suppress** » : access_check, discriminant_check, index_check, length_check, range_check, division_check, overflow_check. Mais comme je doute que vous ayez souvent l'occasion de les mettre en œuvre, je ne m'étendrai pas plus longtemps sur cette notion (la plupart de ces noms évoquent d'ailleurs clairement le test effectué).

Programmation par contrats (Ada2012 seulement)



Ce qui suit ne concerne que les détenteurs d'un compilateur compatible avec la norme Ada2012. Si vous disposez d'une version de GNAT antérieure, cela ne fonctionnera pas.

Avec la norme Ada 2012, il est possible d'effectuer automatiquement des assertions à l'entrée et à la sortie d'une fonction ou d'une procédure. Ces conditions d'application de la fonction sont appelées **contrats**. Pour bien comprendre, changeons d'exemple. Prenons une fonction qui effectue la division de deux **natural** :

Code : Ada

```
function division(a,b : natural) return natural ;
```

Cette fonction doit remplir quelques conditions. Une **précondition** tout d'abord : le nombre b ne doit pas valoir 0 (je rappelle qu'il est impossible de diviser par zéro ). Nous allons donc affecter un contrat à notre fonction :

Code : Ada

```
function division(a,b : natural) return natural
  with Pre => (b /= 0) ;
```

Une exception sera donc levée ou GNAT vous avertira si jamais vous veniez à diviser par 0. Mais notre fonction doit également remplir une **postcondition** : le résultat obtenu doit être plus petit que le paramètre a (c'est une division entière). Nous allons donc affecter un second contrat :

Code : Ada

```
function division(a,b : natural) return natural
  with Pre => (b /= 0) ,
  with Post => (division(a,b) < a) ;
```

Cette programmation par contrat est une des grandes nouveautés de la norme Ada2012, empruntée à des langages tels Eiffel ou D. Elle peut également s'appliquer aux types. Prenons l'exemple d'un type T_Date dont vous souhaitez qu'il soit et demeure valide (pas de 35/87/1901 par exemple). Vous définirez alors un invariant de type :

Code : Ada

```
type T_Date is record
  jour,mois,annee : integer ;
end record
with Type_Invariant => (jour in 1..31 and mois in 1..12 and annee in
  1900..2100) ;
```

Ces nouveautés apportées par la norme Ada2012 vous permettront de mieux sécuriser votre code et de faire reposer une partie du travail de débogage sur le compilateur. Vous savez désormais que le langage Ada est réputé pour sa fiabilité : ses compilateurs ne laissent rien passer ce qui en fait le langage de prédilection d'industries de pointe comme l'aéronautique, l'aérospatiale ou le nucléaire. La programmation par contrats vient ainsi renforcer ces caractéristiques. Plus que jamais, « *avec Ada, si ça compile, c'est que ça marche* ».

En résumé :

- Lorsqu'une erreur est détectée à l'exécution du programme, celui-ci lève une **EXCEPTION** qui met fin au sous-programme en cours. Il vous est toutefois possible de capturer cette exception et d'y apporter un traitement en ajoutant un bloc « **EXCEPTION WHEN** » à la fin de votre programme.
- L'instruction **RAISE** vous permettra de lever vous-même une exception. Dans le bloc de traitement des exceptions d'un sous-programme, l'instruction **RAISE** répercutera au programme appelant le traitement de l'exception.
- Si vous souhaitez traiter immédiatement une exception et rétablir le bon fonctionnement de votre programme, faites appels à l'instruction **GOTO** afin de retenir l'opération ayant échoué.
- Le traitement simple des exceptions peut être accéléré en faisant appel à la directive de compilateur « **PRAGMA Assert()** » ou, avec la norme Ada2012, en utilisant la **programmation par contrats** (préconditions, postconditions et invariants de type).

Multitasking

Après la Programmation Orientée Objet, nous en venons au second point essentiel de la partie IV : le **multitasking**. Si vous n'avez jamais entendu parler de multitasking, peut-être avez-vous déjà entendu parler de **multithreading**, de **parallelisme**, de **programmation multi-tâche**, de **programmation concurrente** ou de **programmation temps réel** ? Non ? Alors sachez que le but de ce chapitre est de réaliser un programme capable d'effectuer plusieurs tâches «en même temps» et non plus successivement, c'est-à-dire de réaliser ses tâches en parallèle et non en série.

Contrairement à la POO, qui consistait avant tout à changer l'organisation de notre code pour mieux le structurer, la programmation temps réel va vous obliger à changer votre façon de programmer et penser vos programmes en y faisant intervenir la notion de temps et en vous obligeant à anticiper les interactions possibles entre les différentes tâches. Il vous faudra également comprendre le fonctionnement du processeur. Attention, ce chapitre est très dense et complexe, ce qui nécessitera peut-être plusieurs lectures.

Parallelisme, tâches et types tâches

Multitasking ou l'art de faire plusieurs tâches à la fois

 Quel est l'intérêt de ce multitasking ? J'ai jamais vu de programmes qui faisaient plusieurs choses différentes «en même temps».

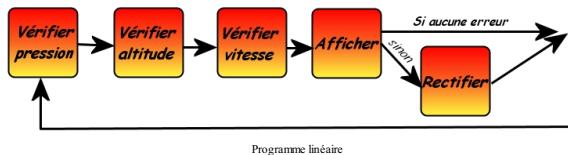
Détrompez-vous ! En ce moment même, vous utilisez assurément un programme multi-tâche : votre système d'exploitation ! Que vous utilisiez Windows, Mac OS ou une distribution Linux, votre système d'exploitation est bien obligé de pouvoir gérer plusieurs tâches en même temps : même si vous lancez un jeu, un navigateur internet et une messagerie mail, votre système d'exploitation n'arrête pas pour autant de fonctionner, de se mettre à jour ou de supporter votre antivirus. Et il effectue ces tâches «en même temps». Le multitasking est donc un composant essentiel de l'informatique moderne.

 Bon, pour le système d'exploitation d'accord. Mais à mon niveau, je ne vais pas programmer un OS ! Quel intérêt de faire plusieurs choses en même temps ? Autant les faire successivement, ça reviendra au même, non ?

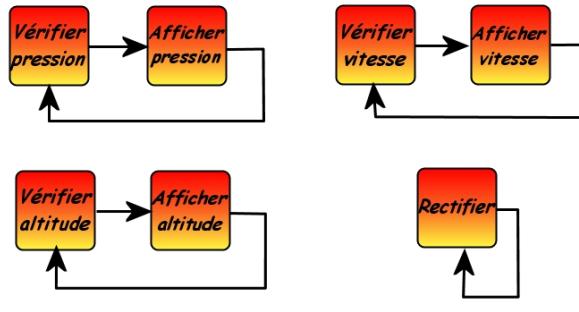


La fusée de Zozor

Prenez comme exemple un domaine très friand de programmation en temps réel : l'aéronautique ! Vous devez rédiger un programme d'aide au pilotage pour la fameuse fusée Z123 pilotée par Zozor. Votre programme doit régulièrement contrôler les capteurs de pression, d'altitude, de température, de vitesse... afficher les valeurs lues et permettre à Zozor de rectifier ces paramètres pour éviter le crash ou l'explosion en plein vol. Si nous procédons comme d'habitude nous aurons le schéma suivant :



Logique n'est-ce pas ? Sauf que le temps que votre programme vérifie la pression, l'altitude ou la vitesse peuvent chuter. Et Zozor devra de plus valider la pression alors que l'urgence serait plutôt de redresser la fusée et de mettre les gaz. Un schéma similaire à celui-ci sera préférable :



Programmes parallèles

Le programme contrôle tous les capteurs en même temps, affiche les valeurs en temps réel et permet de rectifier certains paramètres sans bloquer le reste du système. C'est la réussite assurée pour la Z123.

 Eh bien, dans ce cas, il n'y a qu'à réaliser plusieurs programmes, un pour la pression, un autre pour la vitesse, un autre pour l'altitude...

Ce pourrait être une bonne idée, mais encore une fois, ce n'est pas réaliste : si Zozor rectifie sa trajectoire en augmentant l'altitude, cela impliquera une augmentation de la vitesse et surtout, le système ne devra pas s'affoler de voir la pression atmosphérique ou la température extérieure baisser. Bref, ces différentes tâches manipulent des variables communes et il est donc préférable de les rassembler dans un même programme plutôt que de les dissocier.

Les tâches en Ada

Créer une tâche

Vous avez compris le principe et l'objectif ? Bien, passons alors à la mise en œuvre. Une tâche se déclare en Ada avec le mot clé **TASK** et fonctionne un peu comme une procédure ou une fonction. Cependant, vous devrez préalablement définir sa spécification avant de rédiger son corps. Continuons avec l'exemple de la fusée Z123 :

Code : Ada

```

with ada.text_io ;      use ada.text_io ;
procedure Z123 is
    altitude, vitesse : integer := 0 ;
    --Nous déclarerons les tâches ici
begin
    --La procédure principale se chargera de simuler
    le décollage
    put_line("3 ... 2 ... 1 ... Décollage !");
    while altitude < 10_000 loop
        if altitude < 1000
            then vitesse := vitesse + 5 ;
            altitude := altitude + 5 ;
        else altitude := altitude + 10 ;
        end if ;
    end loop ;
    put_line("Décollage réussi !");
end Z123 ;

```

La procédure Z123 est notre première tâche : la tâche principale. Elle se charge d'augmenter l'altitude et la vitesse au fur et à mesure. Voici maintenant comment nous allons déclarer deux tâches chargées de vérifier l'altitude et la vitesse de la fusée :

Code : Ada

```

task Verif_Altitude ;
task body Verif_Altitude is
begin
loop
put_line("Altitude de" & integer'image(altitude) & " metres")
end loop;
end Verif_Altitude ;
-----;
task Verif_Vitesse ;
task body Verif_Vitesse is
begin
loop
put_line("Vitesse de" & integer'image(vitesse) & " km / H");
end loop;
end Verif_Vitesse ;

```

Vous pouvez remarquer que ces tâches sont déclarées à la façon des packages : d'abord la spécification avec le mot-clé **TASK** puis le corps avec les mots **TASK BODY**. Et d'ailleurs ce dernier ressemble beaucoup à une procédure.



Ces deux tâches commenceront à s'exécuter dès l'instruction **BEGIN** de la procédure Z123.

Avorter une tâche

Compiler et exécuter ce code et là... c'est le drame ! Les lignes d'affichage défilent à une vitesse ahurissante. Aucun moyen de savoir ce qui se passe et le programme n'en finit pas ! Le problème, c'est que la tâche principale ne prendra fin qu'à l'arrêt des deux tâches `Verif_Altitude()` et `Verif_Vitesse()`. Or ces deux tâches ont pour but de vérifier aussi longtemps que possible l'altitude et la vitesse. Il faudrait donc que la procédure principale mette fin à ces deux tâches. Pour cela, il suffit d'utiliser le mot-clé **ABORT** dès que ces tâches ne sont plus nécessaires : on dit alors que l'on fait **avorter** nos tâches :

Code : Ada

```

with ada.text_io ;           use ada.text_io ;
procedure Z123 is
    altitude, vitesse : integer := 0 ;
    task Verif_Altitude ;
    task body Verif_Altitude is
begin
loop
put_line("Altitude de" & integer'image(altitude) & " metres");
end loop;
end Verif_Altitude ;
task Verif_Vitesse ;
task body Verif_Vitesse is
begin
loop
put_line("Vitesse de" & integer'image(vitesse) & " km / H");
end loop;
end Verif_Vitesse ;
begin
put_line("3 ... 2 ... 1 ... Decollage !");
while altitude < 10_000 loop
    if altitude < 1000
        then vitesse := vitesse + 5 ;
        altitude := altitude + 5 ;
    else altitude := altitude + 10 ;
    end if ;
end loop ;
put_line("Decollage réussi !");
abort Verif_Altitude ;
abort Verif_Vitesse ;
end Z123 ;

```

Temporiser vos tâches

Bon, cette fois notre programme prend fin, le décollage de la fusée de Zozor a bien eu lieu mais on n'a le temps de rien voir. Quand le programme prend fin, on voit seulement que l'altitude est de 10000 mètres et la vitesse de 1000 km/H.



Et encore ! Ça ne fait jamais deux fois la même chose ! Des fois, la vitesse ou l'altitude (ou même les deux) valent 0 alors que le décollage est soi-disant réussi ! 😊

Je reviendrai sur ce problème plus tard lorsque nous évoquerons le fonctionnement du processeur. Commençons déjà par régler le problème de la vitesse. Nous allons faire patienter nos deux tâches quelques instants (un dixième de seconde pour être précis) avant un affichage. Cela nous laissera le temps de lire les informations. Pour cela, nous allons utiliser l'instruction **DELAY**. Celle-ci prend en paramètre un nombre décimal à virgule fixe correspondant au nombre de secondes d'attente :

Code : Ada

```

task Verif_Altitude ;
task body Verif_Altitude is
begin
loop
delay 0.1 ;
put_line("Altitude de" & integer'image(altitude) & " metres");
end loop;
end Verif_Altitude ;
task Verif_Vitesse ;
task body Verif_Vitesse is
begin
loop
delay 0.1 ;
put_line("Vitesse de" & integer'image(vitesse) & " km / H");
end loop;
end Verif_Vitesse ;

```

Et pour éviter que le décollage ne soit terminé avant que le moindre affichage n'ait eu lieu, nous allons également temporiser le décollage en attendant un centième de seconde avant l'incrémentation :

Code : Ada - Z123.adb

```

while altitude < 10_000 loop
delay 0.01 ;
if altitude < 1000
    then vitesse := vitesse + 5 ;
    altitude := altitude + 5 ;
else altitude := altitude + 10 ;
end if ;
end loop ;

```

Cette fois, tout se passe comme prévu : la vitesse augmente jusqu'à stagner à 1000 km/H, l'altitude augmente jusqu'à atteindre 10 000 mètres, mettant alors fin au programme de décollage.

Temporiser jusqu'à...

Une autre possibilité est d'utiliser la combinaison de deux instructions : « **DELAY UNTIL ### ;** » où **###** est de type Time. Cela signifie « attendre jusqu'à une date précise ». Vous aurez alors besoin soit du package Ada.calendar soit de Ada.Real_Time. Tous deux définissent un type Time et une fonction `clock` vous permettant d'obtenir la date actuelle. Ainsi, vous pourrez

définir une constante `Delai_Attente` de type `duration` et l'utiliser pour temporiser vos programmes. Reprenons par exemple la tâche `Verif_Vitesse`:

Code : Ada

```
task body Verif_Vitesse is
  Delai_Attente : Duration := 0.1 ;
begin
  loop
    delay until (clock + Delai_Attente) ;
    put_line("Vitesse de" & integer'image(vitesse) & " km / H") ;
  end loop ;
end Verif_Vitesse ;
```

Le principe est assez similaire. Au lieu d'attendre qu'un certain temps se soit écoulé, on attend un moment précis (une heure précise, un jour précis...). Vue la structure actuelle de notre code, vous comprenez que cette solution n'est pas la mieux adaptée à notre problème.

Le type tâche

C'est sympa, mais c'est pas un peu redondant d'écrire deux tâches aussi similaires ? Et puis tu n'as pas expliqué le problème d'affichage d'altitude.

Vous allez devoir attendre encore un peu pour comprendre cette bizarrerie. En revanche, je peux vous éviter ce problème de redondance. Nous allons pour cela définir un type tâche. Nous allons en profiter au passage pour essayer d'améliorer l'affichage et arriver à ceci :

Code : Console

```
3 ... 2 ... 1 ... Décollage !
Altitude (en m)           Vitesse (en km/H)
  8790                      800
```

Ce sera le retour du package `NT_Console` que nous avions utilisé lors du TP sur le jeu du serpent. Je vous préviens tout de suite : vous allez de nouveau être confronté à une bizarrerie que je n'expliquerai pas tout de suite (comment ça je suis devant la difficulté ?). Nous allons donc définir qu'un seul type de tâche appelé `Task_Verif`. Celle-ci affichera le contenu d'une variable à un endroit précis. Il nous suffira ensuite de définir deux tâches `Verif_Altitude` et `Verif_Vitesse` de type `Task_Verif`. Voici comment déclarer notre type tâche :

Code : Ada

```
task type Task_Verif(var : integer ; x : X_Pos ; y : Y_Pos) ;

task body Task_Verif is
begin
  loop
    delay 0.1 ;
    GOTO_XY(x,y) ;
    put(var) ;
  end loop ;
end Task_Verif ;
```

Notez bien deux choses : l'ajout du mot `TYPE` lors de la spécification tout d'abord, l'absence de paramètres pour le corps ensuite. Et voici ce que devient notre programme Z123 :

Code : Ada

```
with ada.text_io ;           use ada.text_io ;
with ada.integer_text_io ;   use ada.integer_text_io ;
with NT_Console ;           use NT_Console ;

procedure Z123 is
  -- Déclaration du type Task_Verif
  altitude : integer := 0 ;
  Verif_Altitude : Task_Verif(altitude,0,2) ;
  vitesse : integer := 0 ;
  Verif_Vitesse : Task_Verif(vitesse,25,2) ;

begin
  GOTO_XY(0,0) ; put("3 ... 2 ... 1 ... Décollage !") ;
  GOTO_XY(0,1) ; put("Altitude (en m) ") ;
  GOTO_XY(25,1) ; put("Vitesse (en km/H) ") ;
  while altitude < 10_000 loop
    delay 0.01 ;
    if altitude > 1000
      then vitesse := vitesse + 5 ;
            altitude := altitude + 5 ;
      else altitude := altitude + 10 ;
    end if ;
  end loop ;
  GOTO_XY(0,3) ; put_line("Décollage réussi !");
  abort Verif_Altitude ;
  abort Verif_Vitesse ;
end Z123 ;
```

En procédant de la sorte, nous nous épargnons toute redondance et il est possible de créer, rapidement et facilement, une tâche pour contrôler la pression, la température ou tout autre paramètre.

Euh... la vitesse et l'altitude n'augmentent plus, c'est normal ?

Oui, c'est normal ! Car j'ai rédigé ce code pour attirer votre attention sur un point particulier : une tâche n'est pas une procédure ou une fonction que l'on appelleait de manière régulière. Elle n'a pas de paramètres en modes `IN OUT` ou `IN OUT`. Je sais vous vous dites : « Pourtant, on a bien initialisé `Verif_Altitude` avec la variable `altitude` ! ». Mais cette variable n'a servi qu'à l'initialisation : la variable altitude valait 0 lorsque l'on a initialisé `Verif_Altitude`, donc son paramètre val vaudra 0, un point c'est tout. La modification de altitude ne modifiera pas pour autant le paramètre val. Comment résoudre ce problème ? Eh bien, il y a toujours les pointeurs !

Code : Ada

```
with ada.text_io ;           use ada.text_io ;
with ada.integer_text_io ;   use ada.integer_text_io ;
with NT_Console ;           use NT_Console ;

procedure Z123 is
  task type Verif(var : access integer ; x : X_Pos ; y : Y_Pos) ;

  task body Verif is
  begin
    loop
      delay 0.1 ;
      GOTO_XY(x,y) ;
      put(var.all) ;
    end loop ;
  end Verif ;

  altitude : aliased integer := 0 ;
  Verif_Altitude : Verif(altitude'access,0,2) ;
  vitesse : aliased integer := 0 ;
  Verif_Vitesse : Verif(vitesse'access,25,2) ;

begin
  ...
```

Le pointeur ne change jamais de valeur, il contient toujours l'adresse de sa cible. Ainsi nous réglons ce problème. Mais il en reste encore un : l'affichage de l'altitude et de la vitesse est plutôt... bizarre. Au lieu d'afficher deux valeurs, le programme en affiche 4 !

Pourquoi ? C'est ce que nous allons expliquer dans la prochaine partie.

Communication inter-tâche directe Le parallélisme, comment ça marche ?

Round-Robin ou «l'effet tourniquet»

Pour comprendre d'où viennent les bizarries rencontrées lors de ces quelques essais, vous devez comprendre qu'un ordinateur n'est pas plus intelligent que vous : il ne peut pas faire deux choses en même temps. Alors comment fait-il pour donner cette impression ? Eh bien, il va vite.

Plus sérieusement, imaginez que votre ordinateur devra faire tourner votre système d'exploitation (OS), votre navigateur internet, un antivirus et un jeu de foot. Il ne peut tout faire en même temps, il va donc exécuter l'OS pendant un laps de temps très court. Une fois ce délai écoulé (on parle de **Time Out**), il mettra cette tâche en mémoire pour s'occuper rapidement du navigateur qu'il placera à son tour en mémoire. Il fera de même avec l'antivirus et votre jeu de foot. Que se passe-t-il ensuite ? Eh bien il sort votre OS de la mémoire pour l'exécuter à nouveau durant un court instant avant de le remettre en mémoire et d'en sortir le navigateur internet, etc. Cette méthode est appelée tourniquet ou round-robin en anglais, ou encore balayage cyclique. C'est le principe du dessin animé : en faisant défiler rapidement des images statiques, on fait croire à un mouvement. Nous avons donc l'impression, pauvres humains que nous sommes, que l'ordinateur effectue plusieurs tâches en même temps.



Mais comment l'ordinateur sait-il quelle tâche il doit retirer de la mémoire ?

Sections critiques

Ah d'accord... mais je ne vois pas le rapport avec nos problèmes d'affichage. ☺

C'est simple. Nos tâches sont (mal)traitées par le processeur de la même manière que les processus. Elles peuvent être interrompues à n'importe quel moment pour être remises en fin de file d'attente. Et pendant qu'elles attendent dans leur file, une autre tâche est traitée qui, elle, reprend son travail là où elle s'était arrêtée. L'idéal serait que les tâches s'alternent de la manière suivante (a représente la tâche à traiter, v la vitesse de la fusée) :

Programme principal	Tâche Verif_Altitude	Tâche Verif_Vitesse
v=10 ; a = 10	-	-
-	placement du curseur colonne 0 afficher a = 10	-
-	-	placement du curseur colonne 25 afficher v = 10
v=15 ; a = 20	-	-
-	placement du curseur colonne 0 afficher a = 20	-
-	-	placement du curseur colonne 25 afficher v = 15

Mais cela ne se passe pas toujours aussi bien et l'enchaînement peut alors ressembler à ça :

Programme principal	Tâche Verif_Altitude	Tâche Verif_Vitesse
v=10 ; a = 10	-	-
-	placement du curseur colonne 0	-
-	-	placement du curseur colonne 25 afficher v = 10
v=15 ; a = 20	-	-
-	afficher a = 20 placement du curseur colonne 0 afficher a = 20	-
-	-	placement du curseur colonne 25 afficher v = 15

Quelle importance me direz-vous ? Eh bien le souci qui va se poser, c'est que lorsque Verif_Altitude() va afficher que a vaut 20, le curseur sera au-delà de la colonne 25 ! Il aura été placé en colonne 25, puis déplacé pour afficher v, mais n'aura pas été remis en colonne 0 avant d'afficher a. D'où l'apparition d'un troisième nombre. Un quatrième apparaîtra si ce phénomène se produit avec Verif_Vitesse().

Ce problème ne semble pas si grave, ce n'est jamais que de l'affichage ! Et pourtant, c'est le problème majeur du multitasking : le partage et l'accès aux ressources (ici l'écran). Prenons un autre exemple. Supposons que nous ayons créé un type tâche **Task** Pilotage permettant de régler vitesse et altitude de la fusée Z123. Nous créons deux tâches : Pilote et Copilote. Que va-t-il se passer si le pilote et le copilote décident tous deux même temps de réduire un peu la vitesse ? La Z123 risque fort de freiner en plein ciel. ☺ Et cela n'a rien d'illégique : supposons que le la tâche pilote lise la valeur de la vitesse avant d'être interrompue. La tâche copilote prend le relai et décide de baisser la vitesse avant d'être à son tour interrompue. La tâche pilote reprend les commandes et affiche la valeur-vitesse lue précédemment. Sauf que la vitesse a entre-temps diminué. Et le pilote ne le saura pas. Conclusion : le pilote va malentendre, pensant sa vitesse trop haute, alors que le copilote s'en est déjà chargé. Résultat : «Houston, we've had a problem».

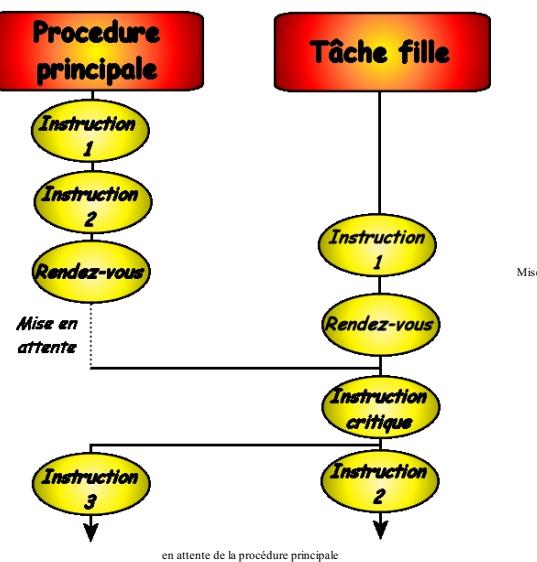
Lorsque vous concevez une tâche, vous devez donc réfléchir à ce que l'on appelle les **sections critiques** (et comme c'est le problème principal, vous DEVEZ ABSOLUMENT y penser). Elles sont faciles à repérer, ce sont les zones de code faisant intervenir des ressources partagées avec d'autres tâches.

Une solution directe : le Rendez-vous

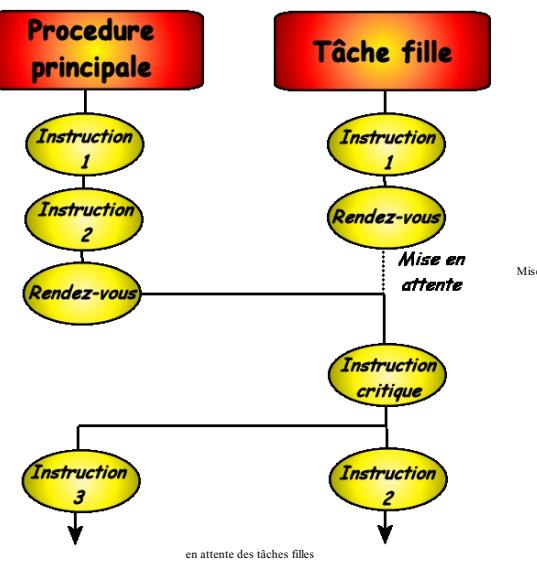
Une première solution pour régler ce problème consiste à faire communiquer nos tâches entre elles directement afin qu'elles se synchronisent. Ainsi, il suffirait que la procédure principale (qui est une tâche elle aussi, je le rappelle), indique aux deux autres à quel moment elles peuvent intervenir sans risquer d'engendrer des comportements erratiques. La procédure principale va donc fixer un **rendez-vous** à ses sous-tâches (le terme anglais est tout simplement Rendez-vous, mais avec l'accent british ☺) : «tu intervientras à tel moment et tu feras telle chose».

Comment cela se traduit-il au niveau du traitement par le processeur ? Deux cas de figures se posent :

- soit la procédure principale est en avance sur les tâches auxquelles elle demande d'intervenir. Auquel cas, la procédure principale sera mise en attente, le temps que les tâches aient terminé leur propres instructions et aient exécuté les instruction demandées par la procédure principale. Après quoi, tout reprendra comme à l'accoutumée.



- soit la procédure principale est en retard sur ses tâches filles. Ces dernières ayant effectué leurs instructions propres seront mises en attente d'un appel venant de la procédure principale.



Les entrées

Les rendez-vous permettent donc à une tâche de recevoir des ordres de l'extérieur. Nous allons pouvoir concevoir des tâches qui ne seront plus hémétiques mais qui seront dotées de points d'entrée. Ainsi, notre type `Task_Verif` étant chargé d'afficher une valeur, nous allons créer une entrée appelée `Affichage`. Cette entrée devra être spécifiée dans la spécification du type tâche et se déclare à l'aide du mot-clé `ENTRY`:

Code : Ada

```

task type Task_Verif(var : access integer ; x : X_Pos ; y : Y_Pos)
is
    entry affichage ;
end Task_Verif ;

```

Cette entrée ne sera en définitive qu'une sorte de sous-procédure attendant le rendez-vous. Attention toutefois, la syntaxe du corps est différente :

Code : Ada

```

task body Task_Verif is
begin
    loop
        accept affichage do
            delay 0.01 ;
            GOTO XY(x,y) ;
            put (var.all) ;
        end affichage ;
    end loop ;
end Task_Verif ;

```

Comprenez cette syntaxe de la manière suivante : «*lorsque la tâche (TASK) accepte (ACCEPT) l'entrée (ENTRY) affichage, elle doit faire (DO) ceci*». Maintenant, si vous lancez votre programme, vous vous rendrez compte que vos deux tâches ne font plus rien, et pour cause, elles attendent qu'un rendez-vous leur soit fixé via l'entrée `Affichage`. Nous allons donc modifier la procédure principale à son tour :

Code : Ada

```

while altitude < 10_000 loop
    delay 0.01 ;
    if altitude < 1000
        then vitesse := vitesse + 5 ;
            altitude := altitude + 5 ;
        else altitude := altitude + 10 ;
    end if ;
    Verif_Vitesse.affichage ;
    Verif_Altitude.affichage ;
end loop ;

```

Complicez, testez : tout fonctionne à merveille ! Les tâches exécutent correctement l'affichage puisqu'elles sont appellées à un moment précis et ne rendent la main qu'après avoir achevé leur entrée. Une remarque toutefois, il aurait été possible de

«déparamétriser» notre type tâche pour paramétrer son entrée. Ce qui aurait évité l'usage d'un pointeur mais nous aurait obligé à spécifier les paramètres au sein de la procédure principale (et en bon fanéant, j'ai préféré la première solution 😊) :

Code : Ada

```
task type Task_Verif is
entry affichage(var : integer ; x : X_Pos ; y : Y_Pos) ;
end Task_Verif ;

task body Task_Verif is
begin
loop
accept affichage(var : integer ; x : X_Pos ; y : Y_Pos) do
  state := var ;
  GOTO_XY(x,y) ;
  put(var) ;
  end affichage ;
end loop ;
end Task_Verif ;
```

Code : Ada

```
while altitude < 10_000 loop
  delay 0.01 ;
  if altitude < 1000
    then vitesse := vitesse + 5 ;
    altitude := altitude + 5 ;
  else altitude := altitude + 10 ;
  end if ;
  Verif_Vitesse.affichage(vitesse,25,2) ;
  Verif_Altitude.affichage(altitude,0,2) ;
end loop ;
```

Les synchronisations sélectives

Améliorons encore nos tâches. Nous allons proposer désormais deux affichages : un normal et un d'urgence qui sera activé lorsque la vitesse dépassera 800 km/h et l'altitude 6000 m. Cette nouvelle entrée appelée simplement `Urgence` affichera la valeur en rouge. La spécification du type tâche est évidente :

Code : Ada

```
task type Task_Verif(var : access integer ; x : X_Pos ; y : Y_Pos)
is
  entry affichage ;
  entry urgence ;
end Task_Verif ;
```



Je veux bien, mais comment faire pour appeler cette entrée ? Si notre tâche attend l'entrée `affichage`, elle n'acceptera pas l'entrée `Urgence` et tout le monde va rester bloqué !

C'est bien vrai, la pire solution serait d'écrire le corps suivant :

Code : Ada

```
task body Task_Verif is
begin
loop
  accept affichage do
    delay 0.01 ;
    GOTO_XY(x,y) ;
    put(var.all) ;
    end affichage ;
  accept urgence do
    delay 0.01 ;
    set_foreground(light_red) ;
    GOTO_XY(x,y) ;
    put(var.all) ;
    set_foreground(gray) ;
    end urgence ;
  end loop ;
end Task_Verif ;
```

Au premier affichage, tout irait bien, mais ensuite, la tâche attendrait nécessairement un affichage d'urgence, ce qui ne risque pas d'arriver au décollage de la fusée. Il faut donc que la tâche propose une alternative, qu'elle se mette en attente d'une ou plusieurs entrées. C'est le rôle de l'instruction `selection`. Nous allons donc écrire :

Code : Ada

```
task body Task_Verif is
begin
loop
select
  accept affichage do
    delay 0.01 ;
    GOTO_XY(x,y) ;
    put(var.all) ;
    end affichage ;
  or
  accept urgence do
    delay 0.01 ;
    set_foreground(light_red) ;
    GOTO_XY(x,y) ;
    put(var.all) ;
    set_foreground(gray) ;
    end urgence ;
  end select ;
  end loop ;
end Task_Verif ;
```

Notre tâche attend donc que l'une ou l'autre des entrées soit appelée et elle sélectionnera alors celle désirée. Il ne reste plus qu'à modifier notre procédure principale :

Code : Ada

```
while altitude < 10_000 loop
  delay 0.01 ;
  if altitude < 1000
    then vitesse := vitesse + 5 ;
    altitude := altitude + 5 ;
  else altitude := altitude + 10 ;
  end if ;
  if vitesse <= 800
  then Verif_Vitesse.affichage ;
  else Verif_Vitesse.urgence ;
  end if ;
  if altitude <= 6000
  then Verif_Altitude.affichage ;
  else Verif_Altitude.urgence ;
  end if ;
end loop ;
```



L'instruction `SELECT` ne doit pas être confondue avec une instruction `IF`, son rôle n'est pas d'orienter vers telle ou telle entrée (quoique) mais bien de proposer d'attendre l'acceptation d'une entrée parmi plusieurs, le choix étant laissé à la tâche appelante. On parle d'attente sélective.

Gardes et compléments

Si l'instruction `SELECT` n'est pas une instruction `IF`, elle peut toutefois imposer certaines conditions : cesser d'attendre ou lever

une exception si l'attente devient trop longue, n'accepter une entrée que sous certaines conditions... Le but recherché n'est pas alors de faire un tri parmi les entrées mais d'éviter des blocages (on parle plus exactement de **faîme**), des levées d'exceptions ou des comportements non désirés. Ces conditions apportées à notre attente sélective sont appelées **gardes**.

Garde conditionnelle

Commençons par imposer une condition toute simple : l'entrée urgence ne pourra être activée que pour une valeur strictement positive : on ne va pas tirer la sonnette d'alarme pour un appareil à l'arrêt. Cela se fera tout simplement avec l'instruction **WHEN** :

Code : Ada

```
task body Task_Verif is
begin
  loop
    select
      accept affichage do
        delay 0.01 ;
        GOTO_XY(x,y) ;
        put(var.all) ;
      end affichage ;
    or when var.all > 0
      accept urgence do
        delay 0.01 ;
        set_foreground(light_red) ;
        GOTO_XY(x,y) ;
        put(var.all) ;
        set_foreground(gray) ;
      end urgence ;
    end select ;
  end loop ;
end Task_Verif ;
```

Garde temporelle



Mais que se passe-t-il si la procédure principale demande à employer l'entrée urgence avec une variable nulle ? Ça ne risque pas de bloquer ?

Ce n'est pas que ça risque de bloquer... ça VA bloquer à coup sûr. Pour éviter tout phénomène d'interblocage, il est possible d'ajouter une garde prenant en compte le temps d'attente : au-delà de 3 secondes d'attente, on exécute une entrée particulière (par exemple, on arrête la tâche proprement). Pour définir le temps d'attente, nous allons réutiliser l'instruction **DELAY**.

Code : Ada

```
task body Task_Verif is
begin
  loop
    select
      accept affichage do
        delay 0.01 ;
        GOTO_XY(x,y) ;
        put(var.all) ;
      end affichage ;
    or when var.all = 0
      accept urgence do
        delay 0.01 ;
        set_foreground(light_red) ;
        GOTO_XY(x,y) ;
        put(var.all) ;
        set_foreground(gray) ;
      end urgence ;
    or delay 3.0 ;
    exit ;
    end select ;
  end loop ;
end Task_Verif ;
```

À ce sujet, il est possible d'utiliser l'instruction **ELSE** plutôt que **OR**. Par exemple si nous écrivons le code ci-dessous (qui déclenchera une exception **TASKING_ERROR**) :



Code : Ada

```
task body Task_Verif is
begin
  loop
    select
      accept affichage do
        delay 0.01 ;
        GOTO_XY(x,y) ;
        put(var.all) ;
      end affichage ;
    or when var.all > 0
      accept urgence do
        delay 0.01 ;
        set_foreground(light_red) ;
        GOTO_XY(x,y) ;
        put(var.all) ;
        set_foreground(gray) ;
      end urgence ;
    else
      exit ;
    end select ;
  end loop ;
end Task_Verif ;
```

Les tâches de type Task_Vérif regarderont si une demande d'affichage ou d'affichage d'urgence a été demandé. Si tel n'est pas le cas, alors (**ELSE**) elles sortiront de la boucle avec **EXIT**. Cela revient à écrire une instruction **OR DELAY 0.0** à la différence qu'avec **ELSE**, le temps durant lequel la tâche est inactive en mémoire n'est pas décompté.

Instruction de terminaison

Plutôt que d'avorter les tâches durant notre programme principal, pourquoi ne pas faire en sorte que celles-ci mettent fin elles-mêmes proprement à leur exécution ? Cela est rendu possible avec l'instruction **TERMINATE**. Je vous conseille donc, à chaque attente sélective, d'ajouter une clause **TERMINATE**, cela évitera que vos programmes n'aient pas de fin parce que vous avez négligé d'avorter toutes vos tâches :

Code : Ada

```
task body Task_Verif is
begin
  loop
    select
      accept affichage do
        delay 0.01 ;
        GOTO_XY(x,y) ;
        put(var.all) ;
      end affichage ;
    or when var.all > 0
      accept urgence do
        delay 0.01 ;
        set_foreground(light_red) ;
        GOTO_XY(x,y) ;
        put(var.all) ;
        set_foreground(gray) ;
      end urgence ;
    or terminate ;
    end select ;
  end loop ;
end Task_Verif ;
```

Communication inter-tâche indirecte Les types protégés

Une seconde façon d'éviter tout problème sur les ressources lié au parallélisme est de s'assurer que l'accès à ces ressources soit protégé et que jamais deux tâches ne puissent y accéder en même temps. Avant de pouvoir lire ou écrire une ressource partagée,

toute tâche devra demander une autorisation au programme principal. Une fois le travail d'écriture ou de lecture effectué, la tâche l'rendra son autorisation, libérant ainsi l'accès à la ressource pour d'autres tâches. Il s'agit du principe d'**exclusion mutuelle**, aussi appelé **Mutex**. La Mutex peut bien entendu être réalisée en Ada, grâce aux types protégés ou **PROTECTED**.

Il s'agit d'un format de type particulier dans lequel il est possible de déclarer les variables protégées, mais aussi des entrées, des fonctions ou des procédures ainsi qu'une partie privée. Les fonctions, procédures et entrées déclarées dans un type protégé sont toutes en exclusion mutuelle. De plus, les entrées peuvent être **bloquantes** car elles doivent être munies d'une garde. Les fonctions seront généralement utilisées pour lire les données protégées, les procédures et entrées pour les écrire.

Pour mieux comprendre, reprenons l'exemple de notre affichage de fusée à la base. Supprimons les entrées et toute communication entre notre programme principal et les tâches. Notre procédure Z123 se résumera donc à ceci :

Code : Ada

```
with ada.text_io ;           use ada.text_io ;
with ada.integer_text_io ;   use ada.integer_text_io ;
with NT_Console ;           use NT_Console ;

procedure Z123 is
    --Nous déclarerons ici notre type protégé--
    -----
    --Nous déclarerons ici notre type tâche--
    -----
```

```
begin
    GOTO_XY(0,0) ; put("3 ... 2 .. 1 .. Decollage !") ;
    GOTO_XY(0,1) ; Put("Altitude (en m)") ;
    GOTO_XY(25,1) ; put("Vitesse (en km/H)") ;

    while altitude < 10_000 loop
        delay 0.01 ;
        if altitude < 1000
            then vitesse := vitesse + 10 ;
            altitude := altitude + 10 ;
        else altitude := altitude + 50 ;
        end if ;
    end loop ;
    abort Verif_Vitesse ;
    abort Verif_Altitude ;
    GOTO_XY(0,3) ; put_line("Decollage réussi !");
end Z123 ;
```

Notre type tâche sera lui aussi simplifié : plus d'attente sélective, plus d'entrées, juste une affichage à un endroit précis :

Code : Ada

```
--notre type tâche
task type Verif(var : access integer ; x : X_Pos ; y : Y_Pos) ;
task body Verif is
begin
loop
    delay 0.01 ;
    GOTO_XY(x,y) ;
    put(var.all) ;
end loop ;
end Verif ;
```

```
--et nos variables
altitude : aliased integer := 0 ;
Verif_Altitude : Verif(alititude'access,0,2) ;
vitesse : aliased integer := 0 ;
Verif_Vitesse : Verif(vitesse'access,25,2) ;
```

Nous allons ensuite pouvoir déclarer un type protégé, que nous appellerons **T_Ecran** et une variable **Ecran** de type **T_Ecran**. Voici un schéma de ce fameux type :

Code : Ada

```
--D'abord les spécifications
protected type T_Ecran(paramètres éventuels) is
    entry truc ;
    procedure bidule ;
    function machin ;
private
    MaVariable : UnTypeDeVariable ;
    --autres déclarations privées
end T_Ecran ;
```

```
--Ensuite le corps du type
protected body T_Ecran is
    entry truc when une_condition is
    begin
        ...
    end truc ;

    procedure bidule is
    begin
        ...
    end bidule ;

    function machin is
    begin
        ...
        return ...
    end machin ;
end T_Ecran ;
```

```
--Enfin, notre objet protégé
Ecran : T_Ecran(paramètres) ;
```



Pourquoi tu n'as pas détaillé ton type **T_Ecran** ? Je dois tout compléter moi-même ?

Non, je compte bien détailler tout cela sans trop tarder, je me contentais juste de vous présenter un plan de la déclaration d'un type protégé. Car il y a plusieurs façons d'utiliser ces types protégés, chaque façon dispose de ses propres avantages et inconvénients, mais aucune n'est parfaite. Je ne compte pas détailler toutes les méthodes possibles et imaginables ; je préfère me concentrer sur deux méthodes parmi les plus connues : les **sémaphores** et les **moniteurs**. La programmation concurrente est un vaste pan de l'informatique, passionnant mais que ce tutoriel n'a pas vocation à couvrir 😊

Les sémaphores

Un peu d'Histoire

Le principe des sémaphores fut inventé par le mathématicien et informaticien néerlandais Edsger Wybe Dijkstra en 1968. Mort en 2002, Dijkstra aura auparavant marqué la seconde moitié du XXème siècle par le rôle qu'il a joué dans le développement de l'informatique en temps que science.



Edsger Wybe Dijkstra. Source: Wikipedia

Inventeur des sémaphores, Dijkstra s'intéressa à la programmation concurrente (nous reviendrons plus tard sur son apport) ainsi qu'à l'algorithme de Dijkstra. Un célèbre algorithme publié en 1959 (notamment étudié en filière économique et social en France) porte son nom : l'algorithme de Dijkstra permet de déterminer un plus court chemin en théorie des graphes (si cela ne vous parle pas, dites-vous que son algorithme permet à votre GPS de connaître la route la plus courte ou la moins chère entre Paris et Lyon).

Il apporta également sa contribution au développement du langage de programmation Algol. Ce langage, très structuré en comparaison des langages de l'époque, permettait la récursivité, donnée par la suite naissance au langage Pascal, langage dont s'inspirera le Français Jean Lehn pour élaborer le langage Ada à la demande du département de la Défense américain. Le langage Ada est donc l'un des héritiers de ce grand Monsieur (ce qui explique en bonne partie sa structure claire et sa rigueur).

De quoi parle-t-on ?

Un sémaphore est, dans la vie courante, un avertisseur. Prenez par exemple des feux de signalisation pour les trains, vous savez, ceux qui font ding-ding-ding... eh bien ce sont des sémaphores. Ils avertissent de la présence ou non d'un train. De même, en espagnol comme en italien, un feu tricolore est appelé sémaphore ou semáforo (sans accent). Bref en informatique, le but d'un sémaphore sera de bloquer les tâches ayant une section critique de manière à n'en laisser passer qu'un certain nombre à la fois. Les sémaphores se chargentront de compter le nombre d'accès restants à une ressource (pour notre écran, seule une tâche pourra y accéder) et de jouer le rôle de feu tricolore. Lorsqu'une tâche arrivera devant une section critique, le sémaphore regardera le nombre de places libres :

- Si ce nombre vaut 0, alors la tâche est mise en attente.
- Sinon, le sémaphore se contentera de décrémenter son compteur de places libres.

Une fois que la tâche aura effectué son travail en section critique, le sémaphore n'aura plus qu'à incrémenter son compteur de places libres afin de libérer l'accès à d'autres tâches en attente. La première opération est généralement notée P. C'est l'initiale du mot *Proberen* qui signifie tester en néerlandais. La seconde opération est généralement notée V pour *Verhogen* (incrémenter). Comme je me doute que, hormis les tulipes et les moulins, vous ne connaissez pas grand chose aux Pays-Bas, voici une petite astuce pour retenir les noms de ces deux opérations : P est également l'initiale de «*Puis-je ?* » ou de Prendre et V l'initiale de «*Vas-y !* » ou Vendre. Compris ? Alors passons à la mise en œuvre.

Mise en œuvre en Ada

Si nous souhaitons réaliser un sémaphore, notre type protégé T_Ecran devra donc contenir une variable privée `compteur` (il serait inutile et dangereux que quelqu'un puisse y accéder sans employer les opérations P et V). Nous l'initialiserons automatiquement à 1 car seulement une seule tâche à la fois pourra accéder à l'écran), mais il est bien sûr possible de l'initialiser via un paramètre. L'opération P devra être réalisée par une entrée pour permettre la mise en attente des tâches. Pour ne pas être mise en attente, il suffira que le compteur n'indique pas zéro places restantes. Il n'est toutefois pas nécessaire que V soit une entrée mais plutôt une simple procédure. En effet, il n'y a aucun besoin de mettre en attente une tâche qui souhaite libérer une ressource et l'opération V n'a besoin d'aucune précondition. Voici donc ce que donnerait notre type protégé :

Code : Ada

```
--les spécifications
protected type T_Ecran is
  entry P ;
  procedure V ;
private
  compteur : integer := 1 ;
end T_Ecran ;

--le corps
protected body T_Ecran is
  entry P when compteur > 0 is      --une condition DOIT être
  spécifiée
  begin
    compteur := compteur - 1 ;
  end P ;

  procedure V is
  begin
    compteur := compteur + 1 ;
  end V ;
end T_Ecran ;
--la variable protégée
Ecran : T_Ecran ;
```

Désormais, nous n'avons plus qu'à revoir le corps de notre type tâche `Task_Verif` afin de positionner notre sémaphore autour de la section critique :

Code : Ada

```
task body Verif is
begin
  loop
    delay 0.01 ;
    Ecran.P ; --Puis-je accéder à la section critique ? Juste un petit
    moment, svp
    GOTO XY(x,y) ;
    put(var.all) ;
  Ecran.V ; --Tu as fini ? Alors Vas-y, sors de la section critique
  et continue !
  end loop ;
end Verif ;
```

Avantages et inconvénients

L'avantage des sémaphores est avant-tout leur simplicité : simple à comprendre et donc simple à mettre en œuvre. Et ce n'est pas un atout négligeable dans un domaine aussi compliqué que la programmation concurrente. Qui plus est, les sémaphores sont peu coûteux en ressources comme le temps processeur.

L'inconvénient est que si un programmeur oublie malencontreusement de libérer une ressource avec l'opération V celle-ci deviendra inaccessible, conduisant à ce que l'on appelle un interblocage. De même, la ressource en elle-même n'est pas protégée, un mauvais programmeur oublierait d'utiliser l'entrée P réalisera un programme sans mutex qui amènera très certainement à de sacrées surprises. Autre inconvénient, la première tâche arrivant au sémaphore s'empare de la ressource partagée et bloque donc les autres : aucune priorité n'est prise en compte. Les sémaphores sont donc un outil simple, pratique mais qui ne répond pas à toutes les attentes. Venons-en donc aux moniteurs.

Les moniteurs

Encore un peu d'Histoire

Les moniteurs, autre outil de synchronisation, furent quant à eux inventés par le britannique Sir Charles Antony Richard Hoare en 1974. Vous avez déjà eu affaire à cet éminent professeur, car il est l'inventeur du fameux *Quick Sort*, l'algorithme de tri rapide que nous avions étudié en début de partie IV avec tant d'autres.



Sir Charles Antony Richard Hoare Source: Wikipedia

Mais ce n'est pas tout. Hoare a également élaboré toute une logique, la logique de Hoare, qui permet de raisonner sur les programmes et algorithmes d'une façon très mathématique, permettant ainsi de traquer d'éventuelles erreurs ou de prouver qu'un algorithme fait effectivement ce qu'il est attendu qu'il fasse.

Eh oui, l'informatique est une science, ne l'oubliez pas. Et comme le disait Hoare : «*Il y a deux manières pour construire un logiciel : ou bien on le fait si simple qu'il n'y a l'évidence pas de défauts, ou bien on le fait si compliqué qu'aucun défaut n'est évident. La première méthode est de loin la plus difficile.*»

Principe du moniteur

Le principe du moniteur est de protéger les ressources partagées en encapsulant les sections critiques de manière à ce que les tâches n'y aient plus accès. On rejoint l'idée de l'**encapsulation** chère à la POO : les données brutes ne sont plus accessibles, on fournit par contre les méthodes pour la manipuler. Mais, toujours pour éviter des accès multiples à cette ressource, on ajoute à ces méthodes quelques conditions, ce que l'on appelle un **verrou de mutex**.

Les moniteurs permettent ainsi une gestion de plus haut niveau que les sémaphores. D'autant plus que le langage Ada, via ses types **PROTECTED** gérera lui-même les mises en attente ou les réveils de tâches. Notre type **T_Ecran**, s'il est conçu tel un moniteur, ressemblera à ceci :

Code : Ada

```
protected type T_Ecran is
    entry affichage(var : access integer ; x : X_Pos ; y : Y_Pos) ;
private
    compteur : integer := 1 ;
end T_Ecran ;

protected body T_Ecran is
    entry affichage(var : access integer ; x : X_Pos ; y : Y_Pos)
    when compteur > 0 is
        begin
            compteur := compteur - 1 ;
            GOTO XY(x,y) ;
            put(var.all) ;
            compteur := compteur + 1 ;
        end affichage ;
    end T_Ecran ;

Ecran : T_Ecran ;
```

Facile n'est-ce pas ? Et attendez de voir ce que devient notre type tâche :

Code : Ada

```
task type Verif(var : access integer ; x : X_Pos ; y : Y_Pos) ;
task body Verif is
begin
    loop
        delay 0.01 ;
        Ecran.affichage(var,x,y) ;
    end loop ;
end Verif ;
```

Cela devient extrêmement simple, non ? Et pourtant les moniteurs sont normalement plus complexes à comprendre que les sémaphores car il faut gérer des signaux de mise en attente ou de réactivation de tâches. Heureusement pour nous, le type **PROTECTED** du langage Ada agit à la façon d'un moniteur ce qui nous décharge de bien des tracas en réglant bon nombre de soucis de sécurité. Et nous pouvons encore améliorer ce code en nous débarrassant de la variable compteur à incrémenter/décrementer. Nous allons modifier la garde de l'entrée **Affichage()** pour ne nous concentrer que sur le nombre de tâches en attente. Nous allons pour cela utiliser l'attribut « **'count** » :

Code : Ada

```
protected type T_Ecran is
    entry affichage(var : access integer ; x : X_Pos ; y : Y_Pos) ;
end T_Ecran ;

protected body T_Ecran is
    entry affichage(var : access integer ; x : X_Pos ; y : Y_Pos) when
        affichage.count = 0 is
        begin
            GOTO XY(x,y) ;
            put(var.all) ;
        end affichage ;
    end T_Ecran ;

Ecran : T_Ecran ;
```

Compléments : priorités et POO

Priorités

Complément sur le tourniquet

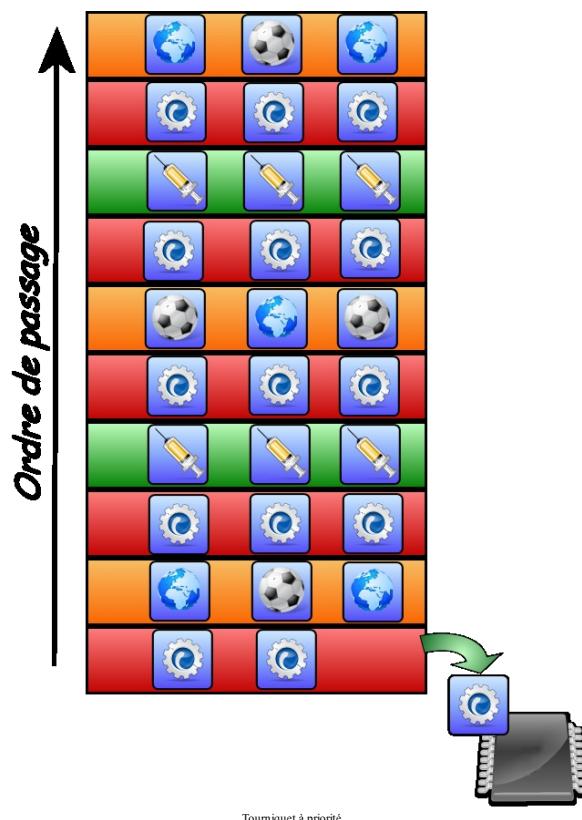
Il est temps désormais de compléter ce que nous avons dit sur le tourniquet. Vous vous souvenez, je vous ai expliqué comment l'ordinateur faisait pour gérer soit-disant «en même temps» un système d'exploitation (OS), un navigateur internet, un antivirus et jeu de foot. Je vous avais expliqué qu'il faisait tourner l'OS pendant quelques millisecondes avant de le placer dans une file d'attente en mémoire, file d'attente gérée sur le mode FIFO (First In First Out). Avant d'avoir de nouveau accès au processeur, votre OS devrait donc attendre que celui-ci ait traité le navigateur internet, puis l'antivirus et enfin le jeu de foot.

Eh bien je vous ai dit des bêtises (pour la bonne cause rassurez-vous). Car il est évident que tous ces logiciels ne fonctionneraient pas sans le système d'exploitation. Celui-ci est primordial au bon fonctionnement de l'ordinateur, contrairement à l'antivirus qui, la plupart du temps, n'a rien de particulier à faire. Il n'est donc pas judicieux que l'antivirus et l'OS aient le même temps d'accès au processeur alors qu'ils n'ont pas la même charge de travail. On dit qu'ils n'ont pas la même **priorité**. C'est pourquoi nos ordinateurs utilisent plutôt un système de **tourniquet à priorité**.



Comment faire pour que le système d'exploitation soit prioritaire puisqu'on utilise une file et donc le principe FIFO ?

L'idée est en fait d'utiliser plusieurs tourniquets et non un seul. On attribue un tourniquet à chaque niveau de priorité. Reprenons notre exemple précédent avec trois niveaux de priorité : 0 pour l'antivirus, 1 pour le jeu de foot et le navigateur internet, 2 pour le système d'exploitation. Les processus de priorité 2 (en rouge sur le schéma) obtiendront 50% du temps processeur, ceux de priorité 1 (en orange) auront 30% et ceux de priorité 0 (en vert) auront 20%. Ce qui nous donne le schéma suivant :



On peut ainsi voir que le système d'exploitation apparaît bien plus souvent que le navigateur internet ou le jeu de foot par exemple.

Un nouveau pragma

La priorité d'une tâche peut être fixée de manière statique grâce à une directive de compilateur. Vous vous souvenez ? Les fameux **PRAGMA**. Nous ferons appel ici au **PRAGMA Priority**. Celui-ci permet de fixer la priorité d'une tâche de manière statique : nous définissons la priorité lors des spécifications de la tâche une fois pour toute. Le langage Ada accepte des niveaux de priorités allant de 0 à 30, 30 étant le niveau de priorité maximal. Nous allons également faire appel à un nouveau package : **System**. Attention ! Pas **Ada.system**, mais **System** tout court ! Ce package nous permet d'utiliser un type **Priority** de type Integer, ainsi qu'une constante **Default_Priority** qui définit une priorité par défaut de 15.

Code : Ada

```
WITH System ; USE System ;
TASK TYPE T_Tache(P : Priority) IS
  PRAGMA Priority(P) ;
END T_Tache ;
TASK BODY T_Tache IS
  ...
END T_Tache ;
```

Le **PRAGMA Priority** peut également être utilisé avec la procédure principale. Par défaut, une tâche hérite de la priorité de la tâche qui l'appelle. C'est-à-dire que si une tâche de priorité 10 déclare des sous-tâches, celles-ci auront une priorité de 10, à moins d'utiliser le **PRAGMA Priority**.



Et il n'est pas possible de modifier la priorité d'une tâche en cours d'exécution ?

Bien sûr que si. Prenez l'exemple dont nous parlions précédemment. L'antivirus n'est pas un processus prioritaire la plupart du temps... sauf lorsque vous tentez d'accéder à une page web ou à un fichier. L'antivirus commence alors son analyse et il est préférable que cette analyse se fasse rapidement afin de ne pas bloquer ou ralentir les processus ayant besoin de ladite page web ou dudit fichier. Pour effectuer tout changement de priorité, vous devrez faire appel au package **Ada.Dynamic_Priorities**. Celui-ci définit deux méthodes dont voici les spécifications simplifiées (pour plus de précision, le package porte le nom **a-dynpri.ads**) :

Code : Ada

```
procedure Set_Priority(P : Priority;
  T : Task_Id := Current_Task);
function Get_Priority(T : Task_Id := Current_Task)
  return Priority;
```

La procédure **Set_Priority** permet de fixer dynamiquement la priorité P de la tâche T (notez que si vous ne renseignez pas le paramètre T, alors **Set_Priority** s'applique par défaut à la tâche qui l'appelle). La fonction **Get_Priority** permet quant à elle de récupérer la priorité d'une tâche (là encore, si le paramètre n'est pas renseigné, c'est la tâche en cours qui est utilisée). Toutefois, n'abusez pas des changements de priorité : ce sont des opérations lourdes. Changer régulièrement la priorité d'une tâche pour l'accélérer peut ainsi favoriser comme conséquence improbable de la ralentir.

Un dernier PRAGMA pour la route

Tant que nous parlons des directives de compilateur, en voici une dernière : le **PRAGMA Storage_Size**. Celui-ci permet de définir la taille (en octets) de l'espace mémoire alloué à une tâche. Par exemple :

Code : Ada

```
TASK TYPE T_Tache(P : Priority) IS
  PRAGMA Storage_Size(500 * 2**10) ;
END T_Tache ;
```

Chaque tâche de type **T_Tache** aura ainsi droit à une taille de 500×2^{10} octets soit 500 ko. Si votre ordinateur ne peut allouer 500 ko lors de la déclaration de la tâche, une exception **STORAGE_ERROR** sera levée.

Quand la programmation orientée objet rejoint la programmation concurrente

INTERFACE, TASK et PROTECTED

Poussons le vice toujours plus loin. Vous avez remarqué que pour appeler l'entrée d'un type protégé, il fallait utiliser une écriture

pointée : `Objet_Protege`.`Entree`. Cela rappelle la POO non ? Eh bien sachez qu'il est possible de concevoir vos tâches ou vos types protégés «façon objet». Ada nous permet ainsi de faire appel aux notions d'héritage ou d'abstraction comme nous le faisions avec des types étiquetés. Pour déclarer un type tâche ou protégé qui puisse être dérivé, nous devrons le définir comme `INTERFACE`, ce qui nous donnera ainsi :

Code : Ada

```
TYPE T_Tache_Mere IS TASK INTERFACE ;
TYPE T_Protege_Pere IS PROTECTED INTERFACE ;
```

L'implémentation se fait comme nous en avons l'habitude :

Code : Ada

```
-- Types interfaces --
TYPE T_Tache_Mere1 IS TASK INTERFACE ;
TYPE T_Tache_Mere2 IS TASK INTERFACE ;

-- Types dérivés --
TYPE T_Tache_Fille1 IS NEW T_Tache_Mere1 AND T_Tache_Mere2 ;
-- OU
TYPE T_Tache_Fille2 IS NEW T_Tache_Mere1 AND T_Tache_Mere2 WITH
... --écrire ici les différentes entrées de la tâche fille
END T_Tache_Fille2 ;
```

Les types tâches filles (mais c'est également valable pour des types protégés fils) héritent ainsi de toutes les méthodes (abstraites nécessairement) s'appliquant aux types tâches mères.

Le type `SYNCHRONIZED`

Mais le langage Ada pousse le vice encore plus loin en termes d'abstraction ! Il est possible de créer un type racine `INTERFACE` sans savoir s'il sera implémenté par un type tâche ou un type protégé. Il s'agit du type `SYNCHRONIZED` ! Un type synchronisé peut soit être une tâche soit un type protégé. Il est donc nécessairement abstrait, et le mot `SYNCHRONIZED` ne s'utilisera donc jamais sans le mot `INTERFACE`.

Code : Ada

```
TYPE T_Synchro IS SYNCHRONIZED INTERFACE ;
```

Il pourra donc être implémenté de différentes façons, permettant par polymorphisme d'utiliser une même méthode pour des tâches ou des types protégés :

Code : Ada

```
TYPE T_Tache_Fille IS NEW T_Tache_Mere AND T_Synchro ;
TYPE T_Protege_Fils IS NEW T_Protege_Pere AND T_Synchro ;
```

Programme multithread et processeur multicœur (Ada 2012 uniquement)

Depuis plusieurs années, les progrès des processeurs ne se font plus seulement sur leur cadence mais surtout sur leur nombre de coeurs. Vous avez sûrement entendu parlé de processeur Dualcore ou Quadcore ? Ils sont en fait composés de deux ou quatre coeurs, c'est à dire de deux ou quatre processeurs. Ces coeurs se répartissent la charge de travail afin de gagner en efficacité, même si un processeur double-coeur ne va pas deux fois plus vite qu'un simple-coeur. Ce nouveau développement des processeurs a ainsi poussé le langage Ada à évoluer. La norme Ada2012 propose ainsi quelques nouveautés dont un package appelé `System.Multiprocessors`. Celui-ci propose ainsi la fonction :

Code : Ada

```
FUNCTION Number_Of_CPUs RETURN CPU ;
```

Cette fonction renvoie le nombre de processeurs disponibles (CPU est l'abréviation de Central Processing Unit, soit Processeur). Mais surtout, vous pouvez attribuer un processeur spécifique à une tâche ou un type tâche lors de sa spécification :

Code : Ada

```
TASK TYPE T_Tache
WITH CPU => 2 ;
```

Ainsi, toutes les tâches de type `T_Tache` seront attribuées au processeur numéro 2. Les coeurs sont numérotés à partir de 1, mais si vous indiquez que CPU vaut 0, alors la tâche pourra être exécutée par n'importe quel cœur.

Exercices fondamentaux

Nous avons jusqu'à là abordé le multitasking sous un angle ludique. Mais tous les cas de parallélisme ne sont pas aussi simples. Pour nous entraîner, nous allons maintenant résoudre trois problèmes classiques de programmation concurrente : le problème des **producteurs et consommateurs**, le problème des **lecteurs et écrivains** et le **diner des philosophes**. Pour information, les deux derniers sont des problèmes qui furent à la fois posés et résolus par Edsger Dijkstra.

Modèle des producteurs et consommateurs

Énoncé

Le modèle producteur-consommateur, ou problème du buffer à capacité limitée, est un cas typique : certaines tâches créent des documents, des données qui sont stockées en mémoire en attendant que d'autres tâches viennent les utiliser. C'est typiquement ce qui se passe avec une imprimante : plusieurs utilisateurs peuvent lancer une impression, les documents sont alors placés dans une file d'attente appelée le spool jusqu'à ce que l'imprimante ait fini son travail en cours et puisse les traiter, les uns après les autres. Le problème qui se pose, c'est que l'espace-mémoire dédié à ce stockage temporaire que l'on nomme **tampon** ou **buffer**, a une capacité limitée et qu'il est en accès partagé.

Et qui dit resource partagée dit problème de synchronisation. Voici typiquement ce qu'il faudrait éviter, on a indiqué par «Time Out» les coupures opérées par le tournequin pour passer d'une tâche à l'autre :

Producteur	Consommateur
Accès au tampon Incrémentation de la taille du tampon (taille vaut 1) Time Out	-
-	Taille = 1 donc accès au tampon Décrémentation de la taille du tampon (taille vaut 0)  Retrait des données
Ajout de données Sortie du tampon Time Out	-

On voit apparaître une erreur grave : lorsque le consommateur accède au tampon, la taille de ce dernier vaut théoriquement 1, sauf que ce dernier est encore vide dans les faits. Le retrait de données entraînera automatiquement une erreur. Nous devrons donc protéger le tampon, et j'ai décidé que nous allions le faire grâce à un moniteur.

Donc votre mission, si vous l'acceptez, sera de créer un type tâche `Producteur` et un type tâche `Consommateur`. Le premier ajoutera dans un conteneur (`doubly_linked_list`, vecteur, file à vous de choisir) des caractères tirés au hasard. Attention, la taille de ce conteneur devra être limitée : imaginez un agriculteur qui rangerait 5000 boîtes de foin dans un hangar prévu pour 2000, ce n'est pas possible. Le buffer est limité ! Le second type de tâche retirera les caractères de votre conteneur, à condition bien sûr qu'il ne soit pas vide. Pour être certain que les caractères ont bien été retirés, chaque consommateur enregistrera son caractère

dans un fichier spécifique. Tout cela ne durera qu'un certain temps, disons quelques secondes. Compris ? Alors à vous de jouer.

Solution

Voici une solution possible au problème des producteurs-consommateurs :

Secret (cliquez pour afficher)

Code : Ada

```

WITH Ada.Text_IO ; USE Ada.Text_IO ;
WITH Ada.Calendar ; USE Ada.Calendar ;
with ada.Containers.Doubly_Linked_Lists ; USE Ada.Containers.Doubly_Linked_Lists ;
WITH Ada.Numerics.Discrete_Random ; USE Ada.Numerics.Discrete_Random ;

procedure prod_cons is
    --PACKAGES

    SUBTYPE Chara IS Character RANGE 'a'..'z' ;
    PACKAGE P_Lists IS NEW Ada.Containers.Doubly_Linked_Lists(Chara) ;
    USE P_Lists ;
    PACKAGE P_Random IS NEW Ada.Numerics.Discrete_Random(Chara) ;
    USE P_Random ;

    --TYPE T_STOCK--
    PROTECTED TYPE T_Stock(Stock_Max_Size : integer := 10) IS
        ENTRY Deposer(C : Character) ;
        ENTRY Retirer(name : string) ;
    PRIVATE
        Stock_Size : Integer := 0 ;
        Stock : List ;
    END T_Stock ;

    PROTECTED BODY T_Stock IS
        ENTRY Deposer(C : Character) WHEN Stock_Size < Stock_Max_Size IS
            BEGIN
                Stock_Size := Stock_Size + 1 ;
                Stock.append(C) ;
            END Deposer ;

        ENTRY Retirer(Name : String) WHEN Stock_Size > 0 IS
            F : File_Type ;
            c : character ;
        BEGIN
            Open(F,Append_File,name) ;
            C := Stock.First_Element ;
            Stock.delete_first ;
            Stock_Size := Stock_Size - 1 ;
            Put(F,c) ;
            Close(F) ;
        END Retirer ;
    END T_Stock ;

    Hangar : T_Stock ;
    --TYPE PRODUCTEUR--

    TASK TYPE T_Prod ;
    TASK BODY T_Prod IS
        G : Generator ;
        c : character ;
    BEGIN
        reset(g) ;
        LOOP
            c := random(g) ;
            Hangar.Deposser(C) ;
            put(c) ;
        END LOOP ;
    END T_Prod ;
    P1, P2 : T_Prod ;
    --TYPE CONSOMMATEUR--

    TASK TYPE T_Cons(nb: integer) ;
    TASK BODY T_Cons IS
        F : File_Type ;
        name : string := "docs/conso" & integer'image(nb) & ".txt" ;
    BEGIN
        Create(F,In_File,name) ;
        Close(F) ;
        LOOP
            Hangar.retirer(name) ;
        END LOOP ;
    END T_Cons ;
    C1 : T_Cons(1) ;
    C2 : T_Cons(2) ;
    C3 : T_Cons(3) ;
    --VARIABLES POUR PROGRAMME PRINCIPAL--
    D : Duration := 4.0 ;
    T : Time := clock ;
    BEGIN
        WHILE Clock < T + D LOOP
            NULL ;
        END LOOP ;
        ABORT P1 ;
        ABORT P2 ;
        ABORT C1 ;
        ABORT C2 ;
        ABORT C3 ;
    end prod_cons ;

```

Modèle des lecteurs et rédacteurs

Énoncé

Second cas typique, le modèle lecteurs/rédacteurs. Vous disposez d'un unique fichier (par exemple un annuaire contenant des numéros de téléphone à 10 chiffres) et plusieurs tâches tentent d'y accéder «en même temps». Certaines pour lire ce fichier (ce sont les lecteurs), d'autres pour y écrire de nouveaux numéros (ce sont les... suspens... rédacteurs). Imaginez les ennuis si un vendeur de téléphone tente de vous attribuer un numéro de portable alors même qu'un de ses collègues à l'autre bout du magasin (ou chez un concurrent) est en train de modifier la base de données. Votre vendeur pourrait vous attribuer un faux numéro composé à partir de deux numéros, ou vous attribuer un numéro que son collègue vient tout juste d'attribuer ! Bref, il faut protéger le fichier.

Les règles sont simples : il est possible à autant de personnes que possible de lire le fichier. En revanche, seule une personne à la fois peut y écrire et bien sûr il est impossible de lire et d'écrire en même temps. De plus, si deux tâches tentent d'accéder au fichier, un lecteur et un rédacteur, alors c'est le rédacteur qui aura la priorité ou, pour être plus clair, il ne devra pas être interrompu par un lecteur.

Ce problème avait été posé et résolu par E. Dijkstra. La solution qu'il proposait faisait appel aux sémaphores, et nous allons donc nous aussi faire appel aux sémaphores. Pour éviter tout problème sur l'accès au fichier, nous aurons besoin de deux entrées P (lecture et P_ecriture), de deux procédures V (lecture et V_ecriture) et donc de deux compteurs (Nb_lecteur et Nb_rédacteur). Cela nous fait ainsi quatre appels possibles. Pour centraliser et clarifier tout cela, je vous propose de ne plus avoir que deux appels (P et V) auxquels nous lournerons en paramètre le type d'accès désiré (lecture ou écriture). Nous utiliserons ainsi un nouveau mot-clé : **REQUEUE**. Celui-ci a pour effet et ré-enfiler la tâche dans la liste d'attente d'une nouvelle tâche. Ainsi, l'entrée P, donnerait quelque chose comme ceci :

Code : Ada

```

ENTRY P(acces : T_acces) IS
BEGIN
    IF Acces = Lecture THEN REQUEUE P_Lecture ;
    ELSE REQUEUE P_Ecriture ;

```

```
END IF;
END P;
```

Attention toutefois, un Time Out peut intervenir une fois cette entrée exécutée. Imaginez qu'une tâche souhaite écrire dans le fichier, l'entrée P redirigea notre rédacteur vers la file d'attente de l'entrée P_Ecriture. Maintenant, si un Time Out intervient au profit d'un lecteur, l'entrée P le redirigea vers P_Lecture, lui permettant ainsi de griller la priorité au rédacteur ! Conclusion : ne vous contentez pas de copier-coller le code ci-dessus.

Votre mission est donc de créer des tâches qui liront un fichier contenant une liste de numéros de téléphone pour en afficher une dizaine à l'écran. Vous créerez également des tâches qui enrichiront le fichier. Restera ensuite à créer un type protégé T_Fichier pour gérer les accès et les priorités. Le code fourni en solution propose un générateur de numéros de téléphone à 10 chiffres.

Solution

Secret (cliquez pour afficher)

Le code proposé protège l'accès au fichier (ce qui était le but de l'exercice) mais pas l'accès à l'écran pour ne pas alourdir la solution.

Code : Ada

```
WITH Ada.Text_IO ;           USE Ada.Text_IO ;
with ada.Numerics.Discrete_Random ;

PROCEDURE Lecteur_Redacteur IS
    SUBTYPE Chiffre IS Integer RANGE 0..9 ;
    PACKAGE P_Random IS NEW Ada.Numerics.Discrete_Random(Chiffre)
    ;
    USE P_Random ;
    g : generator ;
    -----
    -- GÉNÉRATION DU FICHIER --
    -----
    procedure generate_file(name : string) is
        F : File_Type ;
    BEGIN
        Create(F,out_File,"./docs/" & name & ".txt") ;
        reset(G) ;
        FOR J IN 1..20 LOOP
            put(F,'0') ;
            FOR I IN 1..9 LOOP
                put(F,integer'image(Random(G)) (2)) ;
            END LOOP ;
            new_line(F) ;
        end loop ;
        close(F) ;
    END Generate_file ;
    -----
    -- LECTURE-ÉCRITURE DU FICHIER --
    -----
    PROCEDURE Read(Name : String ; line : integer := 1) IS
        F : File_Type ;
    BEGIN
        Open(F,In_File,"./docs/" & Name & ".txt") ;
        IF Line > 1
            THEN Skip_Line(F,Count(Line-1)) ;
        END IF;
        Put_Line(Get_Line(F)) ;
        close(F) ;
    END Read ;
    PROCEDURE Write(Name : String) IS
        F : File_Type ;
    BEGIN
        open(F,append_File,"./docs/" & name & ".txt") ;
        put(F,'0') ;
        FOR I IN 1..9 LOOP
            put(F,integer'image(Random(G)) (2)) ;
        END LOOP ;
        new_line(F) ;
        close(F) ;
    END Write ;
    -----
    -- TYPE PROTÈGE --
    -----
    TYPE T_Acces IS (Lecture,Ecriture) ;
    PROTECTED TYPE T_Fichier IS
        ENTRY P_Acces : T_Acces ;
        PROCEDURE V(Acces : T_Acces) ;
    PRIVATE
        ENTRY P_Lecture ;
        ENTRY P_Ecriture ;
        PROCEDURE V_Lecture ;
        PROCEDURE V_Ecriture ;
        Nb_Lecteur, Nb_Redacteur : Integer := 0 ;
        ecriture_en_cours : boolean := false ;
    END T_Fichier ;
    PROTECTED BODY T_Fichier IS
        ENTRY P(Acces : T_Acces) WHEN not ecriture_en_cours IS
        BEGIN
            IF Acces = Lecture
                THEN REQUEST P_Lecture ;
                ELSE Ecriture_En_Cours := True ;
                    REQUEST P_Ecriture ;
            END IF;
        END P ;
        PROCEDURE V(Acces : T_Acces) IS
        BEGIN
            IF Acces = Lecture
                THEN V_Lecture ;
                ELSE V_Ecriture ;
                    Ecriture_En_Cours := False ;
            END IF;
        END V ;
        ENTRY P_Lecture WHEN Nb_Redacteur = 0 IS
        BEGIN
            Nb_Lecteur := Nb_Lecteur + 1 ;
        END P_Lecture ;
        ENTRY P_Ecriture WHEN Nb_Redacteur = 0 AND Nb_Lecteur = 0 IS
        BEGIN
            Nb_Redacteur := 1 ;
        END P_Ecriture ;
        PROCEDURE V_Lecture IS
        BEGIN
            Nb_Lecteur := Nb_Lecteur - 1 ;
        END V_Lecture ;
        PROCEDURE V_Ecriture IS
        BEGIN
            Nb_Redacteur := 0 ;
        END V_Ecriture ;
    END T_Fichier ;
    Fichier : T_Fichier ;
    File_Name : String := "numeros" ;
    -----
    -- TYPES TÂCHES --
    -----
    TASK TYPE T_Lecteur ;
    TASK BODY T_Lecteur IS
    BEGIN
        FOR I IN 1..10 LOOP

```

```

Fichier_P(lecture) ;
DELAY 0.5 ;
Read(File_Name,I) ;
Fichier_V(lecture) ;
END LOOP ;
END T_Lecteur ;

L1,L2 : T_Lecteur ;

TASK TYPE T_Redacteur ;
TASK BODY T_Redacteur IS
BEGIN
FOR I IN 1..10 LOOP
Fichier_P(ecriture) ;
Write(File_Name) ;
Fichier_V(ecriture) ;
END LOOP ;
END T_Redacteur ;

R1, R2 : T_Redacteur ;

BEGIN
--generate_file(File_Name) ;
null ;
END Lecteur_Redacteur ;

```

Le dîner des philosophes

Énoncé

Le dernier problème, le dîner des philosophes, est un grand classique de programmation concurrente. Inventé lui aussi par Edsger Dijkstra, ce problème d'algorithmique met en avant les problèmes d'interblocage et d'ordonnancement. Le problème est le suivant : cinq philosophes se réunissent autour d'une table ronde pour manger et discuter. Chacun dispose d'une assiette de spaghettis et d'une fourchette située à gauche de la assiette. Mais pour manger, un philosophe doit disposer de deux fourchettes, il doit donc en emprunter une à son voisin de droite. La situation est problématique mais heureusement les philosophes ont également pour habitude de penser, laissant ainsi du temps aux autres pour manger. Un philosophe peut exécuter quatre actions :

- Penser durant un temps indéterminé.
- Prendre une fourchette.
- Rendre une fourchette.
- Manger, là encore durant un temps indéterminé.

Le but du jeu est de faire en sorte que le maximum de philosophes puisse manger en même temps et qu'il n'y ait pas d'interblocage. Exemple simple d'interblocage : si chaque philosophe prend sa fourchette gauche, aucun ne pourra prendre de fourchette droite et tous mourront de faim. Ce genre de situation peut être mis en évidence en insérant régulièrement des instructions `DELAY` dans votre algorithme.

Notre programme ne tournera que durant un temps imparti, disons 10 secondes. Et pour nous assurer que nos philosophes ne meurent pas de faim, nous enregistrerons leurs temps de réflexion et de repas dans un fichier texte : un philosophe qui n'aurait réfléchi que 0,4 secondes et mangé 0,8 secondes a très certainement été confronté à un phénomène de famine : tous les couverts étaient pris et notre philosophe est mort précédemment. D'ailleurs, il serait bon que les temps de réflexion ou de repas soient aléatoires mais ne dépassent pas la seconde.

 La solution consistant à mesurer le temps d'attente et à libérer la fourchette gauche si l'on ne parvient pas à obtenir la fourchette droite en temps voulu est particulièrement mauvaise. Il faut donc concevoir efficacement votre algorithme.

Solution

Secret (cliquez pour afficher)

Code : Ada

```

WITH Ada.Text_IO ; USE Ada.Text_IO ;
WITH Ada.Calendar ; USE Ada.Calendar ;
WITH Ada.Numerics.Discrete_Random ;

PROCEDURE Diner_Des_Philosophes IS
-----  

-- ALÉATOIRE --
-----  

SUBTYPE T_Duree IS integer RANGE 0..1000 ;
PACKAGE P_Random IS NEW Ada.Numerics.discrete_Random(T_Duree)
;  

USE P_Random ;
g : generator ;
-----  

-- CONSTANTES DU PROGRAMME --
-----  

type Philosophe_Nome is (PLATON, SOCRATE, ARISTOTE, EPICURE,
PYTHAGORE) ; --les noms de nos philosophes
Nb_Philosophes : CONSTANT Natural :=  

Philosophe_Nome'Pos(Philosophe_Nome'Last) + 1 ; --le nombre de  

philosophes  

--T_Index est l'intervalle 0..4 pour les différents  

indices --il s'agit d'un type modulaire pour faciliter le calcul  

de l'indice suivant
type T_Index is mod Nb_Philosophes ;
Type T_Boolean_Array is array(T_Index) of Boolean ;
-----  

-- TYPE FOURCHETTE --
-----  

PROTECTED TYPE T_Fourchette IS
--P : prendre les fourchettes si possible
--Le paramètre est un type de sorte que
--P n'est pas une entrée mais une famille d'entrées
ENTRY P(T_Index) ;
--V : rendre les fourchettes
PROCEDURE V(gauche : T_Index) ;
PRIVATE
Libre : T_Boolean_Array := (OTHERS => True) ;
END T_Fourchette ;
-----  

-----  

PROTECTED BODY T_Fourchette IS
--Notez bien la notation particulière des paramètres !
ENTRY P(for gauche in T_Index) when Libre(Gauche) and
libre(gauche+1) IS
BEGIN
Libre(Gauche) := False ;
Libre(Gauche+1) := False ;
END P ;
PROCEDURE V(gauche : T_Index) IS
BEGIN
Libre(Gauche) := True ;
Libre(Gauche + 1) := True ;
END V ;
END T_Fourchette ;
-----  

Fourchette : T_Fourchette ;
-----  

-- TYPE PHILOSOPHE --
-----  

TASK TYPE T_Philosophe(Numer0: T_Index) ;
-----
```

```

TASK BODY T_Philosophe IS
    Name : constant String := 
        Philosophe'Name'Image(Philosophe'Name'Val(Numéro)) ; --On
        extrait le nom à partir du numéro
        Durée : Duration ;
        --Durée aléatoire de la réflexion ou du repas
        Temps : Duration := 0.0 ;
        --Temps d'activité total
        F : File_Type ;
        --Tout sera enregistré dans un fichier
    BEGIN
        --Préparatifs
        create(F,out_file,"./docs/" & Name & ".txt") ;
        --Boucle principale
        LOOP
            --Période de réflexion du philosophe
            Durée := Duration(Random(G))/1000.0 ;
            Temps := Temps + Durée ;
            put_line(F,name & " pense pendant " &
Duration'image(durée) & "s soit " & Duration'image(Temps) & "s
d'activité.") ;
            DELAY Durée ;
            --Période où le philosophe mange
            Fourchette.P(Numéro) ;
            Durée := Duration(Random(G))/1000.0 ;
            Temps := Temps + Durée ;
            Put_Line(F,Name & " mange pendant " &
Duration'image(durée) & "s soit " & Duration'image(Temps) & "s
d'activité.") ;
            DELAY Durée ;
            Fourchette.V(Numéro) ;
        END LOOP ;
    END T_Philosophe ;
    -----
    --On utilisera un tableau pour gérer les philosophes
    --Mais comme ceux-ci doivent être contraints par un numéro
    --Nous utiliserons un tableau de pointeurs sur philosophes
TYPE T_Philosophe access is access T_Philosophe ;
TYPE T_Philosophe_Array IS ARRAY(T_Index) OF
T_Philosophe Access ;
Philosophe : T_Philosophe_Array ;
    -----
    T : CONSTANT Time := Clock + 10.0 ; --indique le temps
d'exécution du programme
    -----
    -- PROGRAMME PRINCIPAL --
    -----
BEGIN
    Reset(G) ;
    --Création des tâches Philosophes (allocation dynamique)
    FOR I IN T_Index LOOP
        Philosophie(I) := NEW T_Philosophe(I) ;
    END LOOP ;
    --Boucle principale permettant de faire tourner le
programme 10 secondes
    WHILE Clock < T LOOP
        null ;
    END LOOP ;
    --Arrêt des tâches Philosophes
    FOR I IN T_Index LOOP
        abort Philosophie(i).all ;
    END LOOP ;
END Diner_Des_Philosophes ;

```

Commentaires concernant la solution proposée

La solution que je vous propose apporte plusieurs nouveautés. Tout d'abord, au lieu de créer cinq variables tâches distinctes elles ont toutes été réunies dans un tableau.

Mais pourquoi avoir utilisé un tableau de pointeurs sur philosophes au lieu d'un tableau de philosophes ? Tu te compliques la vie !

Vous avez du remarquer que mon type `T_Philosophe` n'est pas contraint, il dépend d'un paramètre (numéro). Or, il est impossible de déclarer un tableau contenant des éléments d'un type non contraint. D'où l'usage des pointeurs (cela devrait vous rappeler les chapitres de POO). Ensuite, mes tâches ont besoin d'un générateur `G` pour s'exécuter. Or, il est préférable que `G` soit réinitialisé avec `reset()` avant que les tâches ne débutent. En utilisant les pointeurs, mes tâches ne commencent qu'à partir du moment où j'écris `Philosophie(I) := NEW T_Philosophe(I)` et non à partir de `BEGIN`. Cela permet de retarder leur démarrage. Notez d'ailleurs que si vous utilisez `Ada.Unchecked_Deallocation` sur l'un de ces pointeurs, cela mettra fin à la tâche pointée, un peu viollement d'ailleurs.

Autre particularité, les fourchettes et les philosophes sont placés autour d'une table RONDE ! Et je rappelle qu'un cercle n'a ni début ni fin. Autrement dit, les numéros des fourchettes et philosophes sont 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1... d'où l'emploi d'un type modulaire pour les numéros. C'est vrai que je n'ai pas beaucoup utilisé ces types jusqu'à présent, mais ils se justifient totalement dans ce cas. Ainsi, il suffit de calculer `gauche + 1` pour connaître l'indice de la fourchette de droite, même si la fourchette de gauche a le numéro 4 !

Troisième remarque : vous avez sûrement noté qu'il est impossible que la garde d'une entrée utilise un paramètre de la même entrée. Impossible d'écrire par exemple : `ENTRY P(n : integer) WHEN n > 0 IS`. Or nous n'allons pas rédiger une entrée `P` pour chaque fourchette ! La solution consiste à créer une **famille d'entrées** en inscrivant en paramètre non pas une variable mais un type :

Code : Ada

```

--Spécifications : on n'inscrit que le type, à la manière des
intervalles pour les tableaux
ENTRY P(Integer) WHEN n > 0 ;
--Corps, on indique l'intervalle dans lequel varie notre variable,
à la manière des boucles
ENTRY P(for n in integer) WHEN n > 0 IS

```

C'est avec ces trois exercices fondamentaux que s'achève notre chapitre sur le multitasking. Il y aurait encore beaucoup à dire car la programmation multithread est un vaste domaine cointenant les avancées en matière de processeur. Mais je préfère cesser mes développements pour ne pas trop m'éloigner de notre sujet (le langage Ada) et ne pas alourdir davantage un chapitre déjà compliqué.

En résumé :

- Les `TASK` sont un atout majeur et ancien d'Ada dont d'autres langages se sont inspirés. Cela permet à votre programme d'adopter une tactique non plus linéaire mais multithread. Systèmes d'exploitation, fichiers partagés... les exemples de multitasking sont nombreux et d'actualité.
- Qui dit parallélisme et ressources partagées, dit problèmes de synchronisation. Il vous faut prendre en compte le déroulement et les interruptions de nombreuses tâches simultanées pour concevoir votre algorithme.
- La synchronisation peut se faire directement à l'aide de rendez-vous (via les `ENTRY`) ou indirectement en protégeant les ressources partagées (via les types `PROTECTED` ou `SYNCHRONIZED`) qui sont justement protégés des interruptions.
- Les types protégés permettent d'encapsuler dans un bloc `PRIVATE` vos informations sensibles et d'embarquer les entrées, procédures ou fonctions qui manipulent ces données, au besoin en faisant patienter votre tâche.
- Il est possible de combiner Programmation orientée objet et Programmation concurrente grâce aux types `SYNCHRONIZED`, introduit par la norme Ada2005. Ce type abstrait peut être implémenté par un type tâche ou un type protégé.

Interfaçage entre Ada et le C

L'un des défauts du langage Ada est, il faut bien le reconnaître, son manque de popularité malgré ses grands atouts. Un langage comme le C a connu un grand succès malgré ses défauts et ses lacunes, lacunes que le C++ ou le Java ont tenté de rectifier par la suite. Cette très forte popularité du langage C et de ses descendants a conduit au développement de très nombreuses bibliothèques tierces (on parlerait de packages en Ada) : pour jouer des vidéos, afficher des images, des boutons, jouer des morceaux de musique... et j'en passe. Et si certaines bibliothèques existent aussi en Ada, on ne peut pas en dire autant de toutes.

Tu veux dire que l'on aurait mieux fait d'apprendre un autre langage ?

Non bien sûr ! Ada est un excellent langage ! Mais il faut lui reconnaître ce défaut. Car c'est en reconnaissant ce défaut que des développeurs ont eu l'idée ingénue de permettre aux développeurs Ada d'utiliser des bibliothèques écrites dans d'autres langages : C, C++, Fortran, Cobol. Dès lors, plus besoin de réapprendre un nouveau langage pour utiliser une bibliothèque intéressante : il suffit de la porter en Ada. On parle alors d'interfaçage ou de portage, ou encore de binding. Nous allons donc découvrir comment porter une bibliothèque C en Ada.

Quelques préparatifs

Avant de nous lancer dans l'interfaçage d'une quelconque librairie écrite en C, nous devons nous préparer, tout d'abord en disposant des logiciels nécessaires, ensuite en apprenant quelques rudiments sur le langage C et enfin en se penchant sur les packages qui seront nécessaires.

Logiciels nécessaires

Pas besoin de télécharger des dizaines de logiciels, nous ne comptons pas devenir des développeurs en C, mais nous aurons toutefois besoin d'un compilateur qui connaît le C. Or notre compilateur actuel s'appelle GNAT, c'est-à-dire : GNU NYU Ada Translator (Traducteur GNU pour Ada du NYU [New York University]). GNAT ne compile que de l'Ada et rien d'autre. Sauf que GNAT fait partie d'un projet plus grand : GNU Compiler Collection, un ensemble de compilateurs libres.

Et alors ? Il va pas compiler du C pour autant ?

Non, mais jetez un œil au répertoire où est installé GNAT (par défaut C:\GNAT sous Windows) et plus précisément, jetez un œil au répertoire ...\\GNAT\\2011\\bin. Ce répertoire contient tout plein de programmes comme GNAT lui-même, GPS (IDE fourni par Adacore) mais aussi g++ le compilateur pour le C++ et surtout **gcc le compilateur pour le C** ! Bref nous avons déjà tout ce qu'il faut.

Hey ! Quand je clique dessus, une fenêtre s'ouvre et se ferme ! Il ne se passe rien !

C'est bien normal, le compilateur n'est utilisable que de deux façons : soit par un IDE, soit en ligne de commande. Nous allons opter pour la seconde solution (si certains souhaitent tout de même disposer d'un IDE prévu pour le C, je les renvoie vers l'excellent cours «Apprenez à programmer en C» de Mathieu Neba). Son utilisation est des plus simples, sous Windows allez dans **Menu Démarrer > Tous les programmes > Accessoires > Invite de commandes (ou cmd)**. Sous Linux, cela dépend de votre distribution... sous Ubuntu, le programme s'appelle «Terminal», sous Kubuntu c'est la «Konsole». Ensuite, les commandes seront les mêmes, il vous suffira de taper :

Code : Console

```
cd chemin/du/repertoire/de/votre/fichier/c
```

Cela «ouvrira» le répertoire pour la console. Puis la compilation s'effectuera de la manière suivante :

Code : Console

```
gcc -c nom_du_fichier.c
```

Ainsi, si je dispose d'un fichier appelée `MonProgramme.c` enregistré sur le bureau, je devrais entrer les commandes suivantes :

Code : Console

```
cd C:\Users\Proprietaire\Bureau
gcc -c MonProgramme.c
```

Je vous invite à relire cette partie lorsque vous aurez besoin de compiler votre premier fichier en C.

Quelques rudiments de C

Encore une fois, je ne vais pas faire de vous des programmeurs-C (le cours de Mathieu Neba le fait très bien). Mais de même que pour lire un package, vous n'avez pas besoin de déchiffrer tout son code source, pour connaître les fonctionnalités fournies par un programme en C, vous n'avez pas besoin de connaître toutes ses subtilités. Nous allons donc voir que quelques correspondances entre l'Ada et le C afin que vous puissiez interfaçer correctement vos futures bibliothèques.

Des variables, de leur type et de leur déclaration

En Ada, vous avez désormais l'habitude d'utiliser les types `Integer`, `Float` et `Character`. Ils ont bien évidemment leur pendant en C et s'appellent respectivement `int`, `float` et `char`. Le type `boolean` n'a pas d'équivalent en C (même s'il en a un en C++), le programmeur C utilise donc généralement un `integer` pour pallier à cela en prenant 1 pour `true` et 0 pour `false`.

Qui plus est, le programmeur C utilise régulièrement le type `double` plutôt que `float`. Ce nouveau type correspond en fait au type `Long_Float` en Ada. Et le langage C dispose également d'un deuxième type d'`Integer` appelé `long`. Quelle différence avec les `int` ? Pas aucune aujourd'hui. Enfin, les variables ne se déclarent pas de la même manière. Rapide comparatif :

Version Ada	Version C
Code : Ada	Code : C
<pre>Variable1 : Integer; Variable2, Variable3 : Float; Variable4 : Character := 'A';</pre>	<pre>int Variable1 ; float Variable2, Variable3 ; char Variable4 = 'A' ;</pre>

Vous aurez remarqué les différences : le type est indiqué au tout début et le(s) nom(s) de(s) variable(s) après. Il n'y a pas de symbole pour séparer le type de sa variable. Le symbole d'affectation de l'Ada (`=`) est un simple signe égal (`=`) en C.

Il y a des similitudes également : chaque déclaration se termine bien par un point virgule et les variables sont séparées par des virgules. Les caractères sont symbolisés de la même manière : entre deux apostrophes.

Des programmes

Le langage C dispose bien entendu de fonctions comme le langage Ada, mais il ne dispose pas de procédures. Ou plutôt, disons que les procédures en C sont des fonctions ne retournant aucun résultat. Un exemple :

Version Ada	Version C
Code : Ada	Code : C
<pre>procedure hello ; procedure plusieursHello(n : integer) ; function Addition(a : integer ; b : integer) return integer is begin return a + b ; end Addition;</pre>	<pre>void hello() ; void plusieursHello(int n) ; int Addition(int a, int b) { return a + b ; }</pre>

Comme vous pouvez le voir, le C n'utilise pas les mots `FUNCTION` ou `PROCEDURE`. Une fonction se déclare à la manière d'une variable : le type (celui du résultat attendu) suivi du nom de la fonction, suivi à son tour de parenthèses quand bien même la


```
PROCEDURE Programme_Ada IS
  PRAGMA Linker_Options("HelloWorld.o");
BEGIN
  HelloWorld;           --Appel de la fonction c HelloWorld()
END Programme_Ada;
```

Cette directive indique au compilateur dans quel fichier objet il devra puiser les informations manquantes.

Importation de la fonction C

 Bah ! 😊 Quand j'essaye de compiler, GNAT ne voit toujours pas la fonction HelloWorld() ! Il m'indique « programme_ada.adb:4:04: "HelloWorld" is undefined ».

Attention, la directive de compilateur et le fichier objet ne jouent pas le rôle d'une clause de contexte ou d'un package. Ce n'est pas aussi simple. GNAT sait maintenant dans quel fichier il doit piécher, mais il ne sait pas ce qu'il doit y trouver, ni comment ce qu'il y trouvera a été écrit (est-ce du C, du C++ ou autre chose encore ?). Pour tout vous dire, il est même impossible d'employer directement la fonction HelloWorld() dans notre programme Ada ; nous devons préalablement lui créer un support, un sous-programme Ada auquel on attribuera le code de HelloWorld().

Pour cela, vous devrez écrire la spécification d'une procédure (équivalant d'une fonction `void`) puis importer dans cette procédure le code de HelloWorld() grâce à un second `PRAGMA`, le « `PRAGMA Import` » :

Code : Ada

```
PROCEDURE Hello_World_Ada ;
  PRAGMA Import(Convention => C,
                 Entity   => Hello_World_Ada,
                 External_Name => "HelloWorld");
```

Le `PRAGMA Import` prend trois paramètres :

- Convention, correspond au langage utilisé ;
- Entity correspond au nom de l'entité Ada servant de support, ici il s'agit de notre procédure Hello_World_Ada () ;
- Enfin, le paramètre External_Name est un `String` correspond au nom de la fonction C correspondante.

Attention, le C est sensible à la casse, vous ne devez pas écrire une minuscule à la place d'une majuscule ! Voici donc notre code complet :

Code : Ada

```
PROCEDURE Programme_Ada IS
  PRAGMA Linker_Options("HelloWorld.o");
  PROCEDURE Hello_World_Ada ;
    PRAGMA Import(Convention => C,
                  Entity   => Hello_World_Ada,
                  External_Name => "HelloWorld");
  BEGIN
    Hello_World_Ada;           --Appel de la fonction c HelloWorld()
  END Programme_Ada;
```



Un `PRAGMA Export` existe également. Il fonctionne de la même façon que le `PRAGMA Import` mais permet d'exporter des variables ou des sous-programmes vers des programmes C.

Notre programme Ada avec la norme 2012

Encore une fois, la nouvelle norme Ada2012 apporte son lot de nouveautés et avec elles quelques obsolescences. Ainsi, ce bon vieux « `PRAGMA Import` » est remplacé par des contrats. Au lieu d'écrire le code ci-dessus, vous n'aurez qu'à écrire :

Code : Ada

```
PROCEDURE Programme_Ada IS
  PRAGMA Linker_Options("HelloWorld.o");
  PROCEDURE Hello_World_Ada
  WITH Import => TRUE, Convention => C, External_Name => "HelloWorld";
  BEGIN
    Hello_World_Ada;           --Appel de la fonction c HelloWorld()
  END Programme_Ada;
```

Il est même possible de simplifier ce code en n'écrivant que ceci :

Code : Ada

```
PROCEDURE Hello_World_Ada
  WITH Import, Convention => C, External_Name => "HelloWorld";
```

Supposons que vous disposez d'une fonction `void` en C et d'une procédure en Ada portant le même nom, par exemple `hello()`, votre travail sera encore simplifié :

Code : Ada

```
PROCEDURE Hello
  WITH Import, Convention => C;
```

Quelques menus problèmes

Maintenant que vous avez interfacé votre premier programme du C vers l'Ada, je vous propose de modifier notre programme en C. Mon objectif sera de mettre en évidence quelques soucis ou difficultés auxquels vous pourriez être confrontés.



Nous utiliserons pour la suite de ce chapitre les techniques des normes Ada95 et Ada2005.

Procédure avec paramètres

Code C

Première modification : nous allons améliorer le code du programme C afin qu'il répète plusieurs fois la phrase « Bonjour ### », où les dièses seront remplacés par une chaîne de caractère quelconque (un nom de préférence) :

Code : C

```
#include <stdio.h>
void HelloWorld(int n, char nom[])
{
  int i;
  for(i=0 ; i< n ; i++)
  {
    printf("Bonjour %s ! \n", nom);
  }
}
```

Code Ada

Vous ne devriez pas avoir beaucoup de difficultés à interfaire ce nouveau programme. Pensez seulement à faire appel aux types que nous avons vus dans le package `Interfaces.C`, et notamment au type `Char_Array`. En effet, s'il ne devrait pas y avoir de soucis à utiliser le type `Integer` au lieu de `Int`, il n'en sera pas de même si vous utilisez le type `String` au lieu de `Char_Array`. Mais je ne vous en dis pas davantage pour l'instant. Voici le code en Ada et le résultat obtenu :

Code : Ada

```
WITH Interfaces.C; USE Interfaces.C;

PROCEDURE Programme_Ada IS
  PRAGMA Linker_Options("HelloWorld.o");
  PROCEDURE Hello_World_Ada(n : int; nom : char_array);
  PRAGMA Import(Convention => C,
    Entity      => Hello_World_Ada,
    External_Name => "HelloWorld");
BEGIN
  Hello_world_ada(4, To_C("Bobby"));
END Programme_Ada;
```

Code : Console

```
Bonjour Bobby !
Bonjour Bobby !
Bonjour Bobby !
Bonjour Bobby !
```

Si vous essayez de retirer la fonction de conversion `To_C()`, votre programme devrait lever une exception `CONSTRAINT_ERROR`. Pour mieux comprendre l'intérêt de ce programme, nous allons le modifier à la marge. Au lieu d'afficher `"Bobby"`, nous allons afficher une variable de type `Char_Array` qui vaudra `"Bobby"`. La nuance est infime et pourtant. Voici notre nouveau code :

Code : Ada

```
...  
  Texte : Char_Array(1..5);  
BEGIN  
  Texte := To_C("Bobby");  
  Hello_world_ada(4, Texte);  
END Programme_Ada;
```

Là encore, notre programme se retrouve face à une exception `CONSTRAINT_ERROR`. Et selon lui, ce serait du à un problème de longueur de chaîne.



Mais pourquoi cette erreur ? Il y a bien 5 caractères dans `Bobby`.

En Ada, oui. Mais en C, la gestion des chaînes de caractères est plus complexe. Tout d'abord, en C, chaque case du tableau est un pointeur sur un caractère. Ensuite, chaque chaîne de caractère comporte un caractère supplémentaire indiquant la fin de la chaîne. Il faut donc une chaîne de longueur 6 pour enregistrer 5 caractères en C. Pour régler ce problème facilement, il vous suffit d'indexer votre `Char_Array` de 0 à 5 au lieu de l'indexer de 1 à 5.

Avec un type structuré

Code C initial

Au lieu d'utiliser deux paramètres, nous souhaiterions n'en utiliser qu'un seul de type structuré. Voici ce que donnerait notre code C :

Code : C

```
#include <stdio.h>

struct t_type{
  long n;
  char nom[5];
};

void HelloWorld(struct t_type x)
{
  int i;
  for(i=0 ; i< (x.n) ; i++)
  {
    printf("Bonjour %s ! \n", x.nom);
  }
}
```

Code Ada

Nous devons commencer par créer l'équivalent d'un type `struct` en Ada et indiquer au compilateur à l'aide d'un nouveau `PRAGMA` que ce type est conforme aux exigences du langage C :

Code : Ada

```
TYPE T_Type IS RECORD
  N : Int;
  Nom : Char_Array(0..5);
END RECORD;
PRAGMA Convention(C,T_Type);
```

Voici enfin ce que donnerait notre code complet :

Code : Ada

```
WITH Interfaces.C; USE Interfaces.C;

PROCEDURE Programme_Ada IS
  PRAGMA Linker_Options("HelloWorld.o");
  TYPE T_Type IS RECORD
    N : Int;
    Nom : Char_Array(0..5);
  END RECORD;
  PRAGMA Convention(C,T_Type);

  PROCEDURE Hello_World_Ada(X : t_type);
  PRAGMA Import(Convention => C,
    Entity      => Hello_World_Ada,
    External_Name => "HelloWorld");
  X : T_Type;
BEGIN
  X.N := 4;
  X.Nom := To_C("Bobby");
  Hello_World_Ada(x);
END Programme_Ada;
```



Mais ? 😐 Comment se fait-il que le message ne cesse de s'afficher en boucle ? Et d'ailleurs, ce n'est même pas le bon message !

Je vous propose cet exemple pour attirer votre attention sur un problème d'interfaçage. Les types `struct` du C correspondent aux types `RECORD` de l'Ada mais le langage requiert que lorsqu'un type `RECORD` est passé en paramètre d'une fonction ou d'un programme, il soit transmis via son adresse, c'est-à-dire grâce à un pointeur.

Résoudre notre problème

Une première solution, un peu longue, est de revoir complètement nos codes. Nous devons d'abord revoir le code en C :

Code : C

```
#include <stdio.h>

struct t_type{
    long n ;
    char nom[5] ;
} ;

void HelloWorld(struct t_type *x)
{
    int i ;
    for(i=0 ; i< (x->n) ; i++){
        printf("Bonjour %s ! \n", x->nom) ;
    }
}
```

Notez bien l'usage des symboles * et ->. Nous utilisons désormais un paramètre X de type pointeur. Vénons-en maintenant à notre code Ada :

Code : Ada

```
WITH Interfaces.C ;      USE Interfaces.C ;

PROCEDURE Programme_Ada IS
    PRAGMA Linker_Options("HelloWorld.o") ;

    TYPE T_Type IS RECORD
        N : Int ;
        Nom : Char_Array(0..5) ;
    END RECORD ;
    PRAGMA Convention(C,T_Type) ;

    PROCEDURE Hello_World_Ada(X : access t_type) ;
    PRAGMA Import(Convention => C,
                  Entity       => Hello_World_Ada,
                  External_Name => "HelloWorld") ;

    X : ALIASED T_Type ;
BEGIN
    X.N := 4 ;
    X.nom := to_c("Bobby") ;
    Hello_World_Ada(X'access) ;
END Programme_Ada ;
```

Notre problème est alors réglé. Le programme retrouve son fonctionnement normal mais un détail me chiffonne : en règle générale, vous serez amenés à interfaçer des bibliothèques importantes et vous ne pourrez pas modifier le code source de chacune des fonctions C afin qu'il corresponde à vos envies. Ce serait un travail long, fastidieux et surtout risqué. L'idéal serait de ne pas toucher au code source écrit en C. Heureusement, Ada a tout prévu ! Pour cela, revenez au code C fourni au début de cette sous-partie (celui ne faisant pas appel aux pointeurs, c'est-à-dire sans les symboles * et ->). Nous allons simplement indiquer que notre type répond à un **PRAGMA Convention** légèrement différent :

Code : Ada

```
WITH Interfaces.C ;      USE Interfaces.C ;

PROCEDURE Programme_Ada IS
    PRAGMA Linker_Options("HelloWorld.o") ;

    TYPE T_Type IS RECORD
        N : Int ;
        Nom : Char_Array(0..5) ;
    END RECORD ;
    PRAGMA Convention(C_pass_by_copy,T_Type) ;

    PROCEDURE Hello_World_Ada(X : t_type) ;
    PRAGMA Import(Convention => C,
                  Entity       => Hello_World_Ada,
                  External_Name => "HelloWorld") ;

    X : T_Type ;
BEGIN
    X.N := 4 ;
    X.nom := to_c("Bobby") ;
    Hello_World_Ada(X) ;
END Programme_Ada ;
```

Comme vous pouvez le voir à la ligne 11, au lieu d'indiquer que T_Type répond aux conventions du langage C, nous indiquons avec **C_Pass_By_Copy** qu'il devra également être copié avant d'être transmis à un sous-programme. Plus besoin de pointeurs, de variables **ALIASED** ou de modification du code source en C. Notre problème est réglé vite fait bien fait.

En résumé :

- Avant d'utiliser du code C dans votre programme Ada, vous devez compiler votre fichier « .c » afin de générer un fichier « .o ».
- Pour indiquer le fichier objet utilisé, faites appel au **PRAGMA Linker_Options**. Attention, seul le fichier objet peut être là !
- Les variables ou sous-programmes peuvent être importé grâce au **PRAGMA Import**. Vous devez alors indiquer le langage choisi, la variable ou sous-programme servant de support en Ada ainsi que son équivalent en C.
- Quelques différences de fond existent entre le C et l'Ada. Pensez qu'une chaîne de caractères, en C, contient toujours un caractère supplémentaire pour indiquer la fin de la chaîne. Quant aux types structurés, ils ne peuvent être passés en paramètre que via un pointeur.

Ça y est ! Vous êtes venus à bout de cette quatrième et infâme partie ! Que de théorie. J'aperçois d'ici la fumée qui se dégage des neurones en surchauffe ! ☺ Vous pouvez désormais passer à la dernière partie (si ce n'est pas déjà fait ☺) sur la programmation événementielle avec GTK. À vous les fenêtres et les boutons, enfin ! Cette cinquième partie sera à l'exact opposé de la quatrième : très pratique, avec des objectifs bien plus palpitants que de connaître le temps d'exécution d'un algorithme de tri, et surtout, c'est le couronnement de tous les efforts fournis pour parvenir jusque-là. Alors régalez-vous !

Partie 5 : Ada et GTK : la programmation évènementielle

Vous en reviez depuis le début : créer de jolies fenêtres avec des boutons, des images, des menus... qui réagissent lorsque vous cliquez avec votre souris et pas seulement lorsque vous tapez au clavier ! Eh bien ça y est ! Vous y êtes ! Nous allons commencer par un chapitre généraliste sur GTK : de quoi s'agit-il ? Pourquoi ce choix ? Comment l'installer ? Comment le faire fonctionner ? Puis, nous entrons dès le second chapitre dans le vif du sujet en créant nos premiers programmes fenêtres.

 À tous ceux qui auraient évité les parties précédentes pour venir directement à celle-ci, je n'aurai qu'une chose à dire : «Lisez au moins les trois premières parties ainsi que le chapitre 5 de la partie IV, sans quoi vous n'allez rien comprendre !»

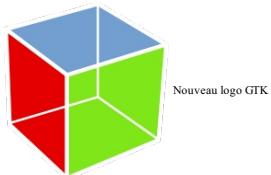
GTKAda : introduction et installation

Ca y est, après des dizaines de cours théoriques faisant appel à une vieille console en noir et blanc, des heures passées à écrire des lignes de code pour n'aboutir qu'à un médiocre logiciel ne reconnaissant que le clavier, nous allons enfin pouvoir nous lancer dans la conception de logiciels plus modernes, avec des images, des boutons qui réagissent quand on clique dessus... Pour faire cela, plusieurs choix s'offraient à nous, j'ai choisi de vous en proposer un : **GTK** et surtout sa version prévue pour Ada, **GTKAda**.

Avant de rentrer dans le vif du sujet, je vais prendre quelques unes de vos précieuses minutes pour vous exposer ce que sont GTK et GTKAda. Je vous expliquerai pourquoi j'ai fait ce choix alors que d'autres solutions s'offraient à nous. Nous verrons également comment l'installer et enfin comment l'utiliser.

Vous avez dit GTK ?

Qu'est-ce que GTK ?



GTK n'est pas un logiciel. Il s'agit d'un ensemble de bibliothèques permettant de créer des interfaces graphiques pour vos programmes. Il existe plusieurs bibliothèques de la sorte, elles permettent aux développeurs de créer rapidement des interfaces graphiques (aussi appelées GUI, pour **Graphical User Interface**), c'est-à-dire des programmes fenêtrés, disposant de boutons, de textes, d'images... et permettant à l'utilisateur d'agir via son clavier mais également via la souris ou d'autres périphériques. Ces bibliothèques fournissent aux programmeurs tout un ensemble de méthodes leur évitant de réinventer la roue : pas question de perdre du temps à dessiner les bords du bouton, à définir sa couleur principale, sa couleur lorsqu'il est survolé, sa couleur lorsqu'il est enfoncé...

 Et GTK dans tout cela ? 

GTK est l'une de ces bibliothèques. C'est l'acronyme pour **Gimp ToolKit**, cette bibliothèque ayant été à l'origine créée pour le puissant logiciel de traitement d'images GIMP. Elle est depuis utilisée dans de nombreux logiciels :

- Le logiciel de retouche d'image Gimp, bien sûr ;
- L'environnement graphique Gnome, utilisé sur de nombreuses distributions Linux, telle la célèbre Ubuntu ;
- Le logiciel de virtualisation VmWare ;
- Le logiciel de dessin vectoriel Inkscape ;
- De nombreux logiciels ayant l'initial G : GEdit, GNumerics, GParted, GThumb... et d'autres encore tels Nautilus ou Totem ;
- Vous pourrez trouver une liste de logiciels réalisés sous GTK en allant sur le site [GTK+ Applications Repository](#).



Gimp



Gnome



Inkscape



VmWare

GTK, GTKAda, GDK, Glib et toute la famille

 C'est quoi la différence entre GTK et GTKAda ? J'ai aussi entendu parler de GDK et GTK+ : c'est mieux ou pas ?

Avant d'aller plus en avant, il faut clarifier tous ces noms et logos. Le projet GTK (aussi appelé GTK+) est une bibliothèque écrite en C et pour le C. Elle est toutefois disponible dans de nombreux autres langages de programmation dont Ada. Mais pour cela, il aura fallu réaliser un interface (ou binding), une traduction partielle de la bibliothèque. GTKAda est donc cette traduction de GTK du C vers l'Ada. Comme de nombreux projets liés à l'univers Linux, GTK est composé de différentes couches (des sous-projets en quelques sortes) :

- **Cairo** : il s'agit d'une bibliothèque graphique 2D. Absente des premières versions de GTK, Cairo prend de plus en plus de place dans le projet.
- **GLib** : il s'agit d'une bibliothèque permettant d'étendre les possibilités du C, et notamment de manipuler diverses structures de données. Elle n'a rien à voir avec le rendu graphique mais est à la base du projet GTK+.

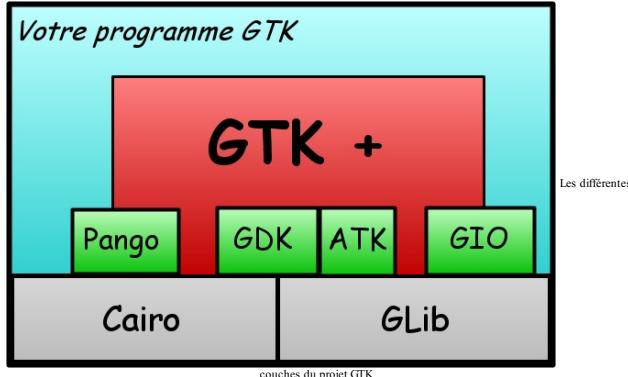


Logo de Cairo

Au-dessus, viennent se greffer d'autres couches :

- **Pango** : cette bibliothèque est en charge du texte. Elle permet l'utilisation de diverses polices, la mise en forme du texte ainsi que l'écriture dans divers alphabets (anglo-saxon, latin, scandinave, cyrillique...). Avec elle, plus de soucis avec les caractères accentués ! vous pourrez même écrire en japonais si le cœur vous en dit.
- **GDK** : pour GIMP Drawing Kit. Il s'agit de la couche graphique bas-niveau de GTK+. Elle était présente dès l'origine de GTK mais tend à céder la place à Cairo.
- **ATK** : ou Accessibility Toolkit. Cette bibliothèque permet une meilleure accessibilité des logiciels aux personnes handicapées : forts contrastes, loupes, navigation au clavier seul ou à la souris seule... nous ne nous servirons pas de cette bibliothèque, mais libre à vous d'y jeter un oeil.
- **GLib** : de moindre importance, cette bibliothèque est désormais intégrée à Glib.

Le projet GTK+ fait appel à ces différentes couches. Ce que l'on peut résumer par le schéma suivant.



Autre acronyme que vous pourriez voir : **Glade**. Il s'agit d'un logiciel permettant la réalisation de votre interface graphique en quelques clics. Nous n'utiliserons pas Glade car il est important que vous maîtrisez GTKAda au préalable afin de comprendre le code qui sera généré.



Pourquoi ce choix ?

?

Pourquoi avoir choisi une bibliothèque écrite en C ?

J'ai avant tout choisi l'une des bibliothèques graphiques les plus **connues**, non pas que ce soit un gage de qualité mais cela vous apporte l'assurance de trouver des ressources complémentaires (tutoriels, cours, forums...). Certes elle est écrite en C, mais elle a bénéficié d'un **interface vers Ada** complet et de qualité. Pour tout vous dire, vous ne verrez presque jamais de C en vous baladant dans les packages. D'ailleurs, GTK n'est pas réservé au C, il est disponible dans de nombreux autres langages : C++, C#, Java, Javascript, Python, Vala et Perl pour les bindings officiels. Il est également supporté (tout ou partie) par Ruby, Pascal, R, Lua, Ocaml, Haskell, FreeBasic, D, Go ou Fortran. Ouf ! impressionnant non ?

?

J'ai plusieurs ordinateurs avec différents systèmes d'exploitation. Je suppose que ça ne marche que sous Windows ton GTK ?

Pas du tout, GTK est **multiplateforme**, vos projets pourront être développés sous Windows, Mac OS, Linux ou Unix. Bien sûr, il faudra recompiler sous chaque système d'exploitation, les .exe ne fonctionnant pas ailleurs que sous Windows (c'est un exemple).

?

Et ça va me coûter combien ?

Pas un centime ! GTK est un projet **gratuit et libre**. Il est placé sous la licence des logiciels libres LGPL. Cela vous permettra, si vous le souhaitez, de vendre votre programme sans pour autant en faire un logiciel sous licence GPL. En revanche, si vous venez à modifier le code source de GTK, alors la bibliothèque obtenue devrait absolument être sous licence LGPL.

?

J'ai regardé sur internet des logiciels faits avec GTK... ils sont moches !

GTK a un aspect très... Linux qui ne plait pas à tous. Et pour cause, l'environnement de Bureau GNOME, utilisé par de nombreuses distributions, est basé sur GTK. Mais rassurez-vous, si cet aspect ne vous plaît pas, GTK est **personnalisable**. Différents thèmes graphiques sont aisément téléchargeables sur le site [Gnome-look.org](http://gnome-look.org).

GTK est donc une bibliothèque libre, gratuite, multilingue et notamment bien adaptée à l'Ada, multiplateforme, personnalisable, très utilisée et pour laquelle vous trouverez donc aisément des informations sur le net. J'ajouterais encore un avantage à GTK : sa conception **orientée objet**. Pour vous qui venez tout juste de découvrir la POO, il me semble intéressant que la bibliothèque que vous allez étudier soit elle aussi orientée objet.

Télécharger et installer GTKAda

[Télécharger](#)

Autre atout de GTKAda : il est aisément téléchargeable sur le site d'Adacore, à l'adresse suivante : <http://libre.adacore.com/download/configurations>. Après avoir vérifié votre système d'exploitation (voir le 1 du schéma ci-dessous), dans le menu **GtkAda 2.24.2 (2)** (la version en cours lorsque je cr��is ces lignes), sélectionnez le bouton **gtkada-gpl-2.24.2-nt-exe (3)** puis cliquez sur le bouton **Download selected files (4)**. Sous Linux, vous aurez deux archives à télécharger : **gtkada-gpl-2.24.2-src.tgz** et **gtk+-2.24.5-i686-pc-linux-gnu.tgz**.



Installer

Sous Windows

Sous Windows, l'installation est elle aussi simplissime. Ouvrez le fichier zip téléchargé ainsi que les sous-répertoires qu'il contient. Extrayez le fichier **gtkada-gpl-2.24.2-nt.exe** puis lancez-le et suivez les instructions.

Sous Linux

Ouvrez le répertoire /usr et créez un sous-répertoire gtkada. Dans l'archive téléchargée, vous trouverez deux sous-archives. Commencez par ouvrir **gtk+-2.24.5-i686-pc-linux-gnu.tgz** et décompresser son contenu dans le répertoire /usr/gtkada que vous venez de créer.

La seconde sous-archive pourra être décompressée dans un répertoire temporaire. À l'aide de la console, placez-vous dans ce répertoire temporaire et entrez les instructions suivantes :

```
Code : Console
configure --prefix=/usr/gtkada
make all
sudo make install
sudo ldconfig
```

Vous pourrez ensuite supprimer le répertoire temporaire.

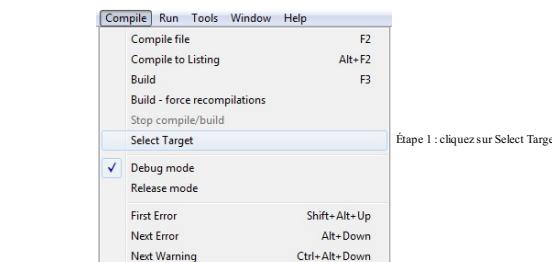
Configurer votre IDE

Attention, ce n'est pas parce que GtkAda est installé que vous pouvez déjà l'utiliser. Vous devrez préalablement configurer votre IDE pour qu'il indique au compilateur où se trouvent les packages de GtkAda.

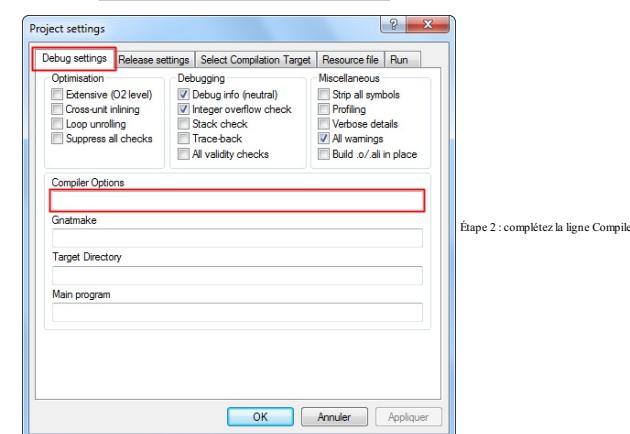
Sous Adagide

Pour Adagide, il vous suffit d'aller dans le menu *Compile > Select Target*. Une fenêtre s'ouvre, cliquez alors sur l'onglet *Debug settings*. Dans la ligne Compiler Options, vous pouvez ajouter -I suivi immédiatement du chemin pour accéder au répertoire GtkAda/include/gtkada. Par exemple, voici ce que vous pourriez écrire (tout dépend du répertoire choisi pour installer GtkAda) :

```
Code : Compiler options
-IC:\Program Files\GtkAda\include\gtkada
```



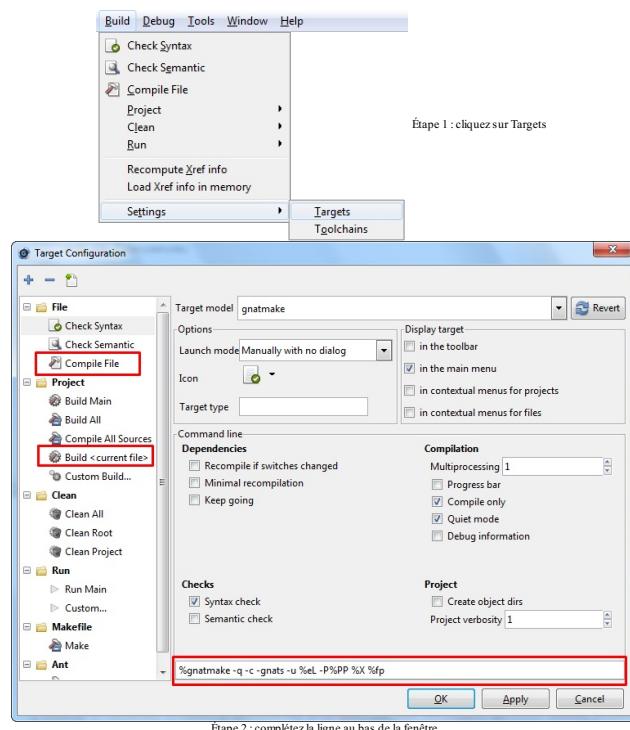
Étape 1 : cliquez sur Select Target



Étape 2 : complétez la ligne Compiler

Options
Sous GPS

Une manipulation similaire est nécessaire sous GPS. Rendez-vous dans le menu *Build > Settings > Targets*. Puis dans chacun des outils que vous utilisez (c'est-à-dire *Compile File* et *Build <current file>*), ajoutez à la ligne d'instruction située au bas de la fenêtre la commande indiquée précédemment.



Un premier essai

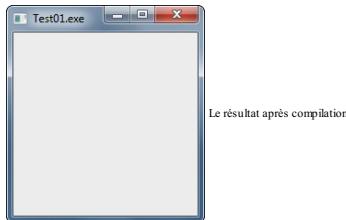
Pour s'assurer que tout fonctionne correctement, copiez le code ci-dessous. N'oubliez pas d'indiquer au compilateur l'emplacement des packages de GtkAda.

Code : Ada

```
WITH Gtk.Main ;      USE Gtk.Main ;
WITH Gtk.Window ;   USE Gtk.Window ;

PROCEDURE Test01 IS
    win : Gtk_Window ;
BEGIN
    Init ;
    Gtk_New(Win) ;
    Win.show_all ;
    Main ;
END Test01 ;
```

Compiler, exécuter et vous devriez obtenir ceci :



Ce n'est pas extraordinaire, mais c'est bien une fenêtre tout ce qu'il y a de plus normale, pas une vilaine console ! Vous venez de réaliser votre tout premier programme fenêtré avec GTK !

Vous devriez voir apparaître un fichier `gnatlogo`. Je vous déconseille de le supprimer car c'est dans ce fichier qu'est inscrit le chemin `-IC:\...\Gtkada\include\gtkada`. Alors si vous ne souhaitez pas le réécrire à chaque fois, ne le supprimez pas.

En résumé :

- GTK est un ensemble de bibliothèques graphiques codées en C, mais le projet GtkAda permet d'utiliser GTK tout en codant en Ada.
- La conception de GTK est orientée objet. Il sera donc possible d'utiliser l'héritage et la dérivation.
- Avant tout projet utilisant GTK, vous devrez absolument indiquer au compilateur le chemin pour accéder aux packages de GTK.

Votre première fenêtre

Vous avez créé votre premier programme fenêtré lors du précédent chapitre... mais sans rien comprendre à ce que vous faisiez. Nous allons donc prendre le temps cette fois de décorner et de comprendre ce premier bout de code GtkAda.

Analysons notre code

Code GtkAda minimal

Pour commencer, nous allons éduquer encore notre code. Voici donc le minimum requis pour tout programme GTK :

Code : Ada

```
WITH Gtk.Main ;      USE Gtk.Main ;
PROCEDURE MaFenetre IS
BEGIN
    Init ;
    Main ;
END MaFenetre ;
```

Vous pouvez compiler, vous vous apercevez qu'il ne se passe pas grand chose : la console s'ouvre comme d'habitude mais plus de jolie fenêtre ! 😱 Et pourtant, GTK a bien été lancé. En fait, comme bon nombre de bibliothèques, GTK nécessite d'être initialisée avant toute utilisation avec la procédure `Init` ou plus exactement `Gtk.Main.Init`. Je vous précise le chemin complet car vous serez très certainement amenés à un moment ou à un autre à le préciser.

Puis, votre code doit se terminer par l'instruction `Main` (ou `Gtk.Main.Main`). Pourquoi ? Nous expliquerons cela en détail lors du chapitre sur les signaux, mais sachez pour l'instant que cette instruction équivaut à ceci :

Code : Ada

```
LOOP
    Traiter_Les_Actions_De_L_Utilisateur ;
END LOOP ;
```

L'instruction `Main` empêche ainsi que votre fenêtre se ferme aussitôt après avoir été créée et se chargera de réagir aux différentes actions de l'utilisateur (comme cliquer sur un bouton, déplacer la souris, appuyer sur une touche du clavier...). `Main` n'est donc rien d'autre qu'une boucle infinie, laquelle peut toutefois être stoppée grâce à la procédure `Main.Quit` que nous verrons à la fin du chapitre et qui fait elle aussi partie du package `Gtk.Main`.

Créer une fenêtre

Une fenêtre sous GTK est un objet **PRIVATE** de la classe `Gtk.Window_Record`. Cette classe et les méthodes qui y sont associées sont accessibles via le package `Gtk.Window`. Cependant, vous ne manipulerez jamais les objets de classe `Gtk.Window_Record`, vous ne manipulerez que des pointeurs sur cette classe.



Augh ! 🤪 Des pointeurs sur des classes privées. Il va falloir que je relise toute la partie IV pour comprendre la suite ?

Ne vous inquiétez pas, tout se fera le plus simplement du monde. Plutôt que de manipuler directement des objets `Gtk.Window_Record`, vous utiliserez des `Gtk.Window`. Le fait qu'il s'agisse d'un pointeur sur classe ne compliquera pas votre développement, mais il est bon que vous sachiez qu'une fenêtre GTK se définit ainsi :

Code : Ada

```
type Gtk_Window is access all Gtk_Window_Record'Class;
```

À quoi bon savoir cela ? Eh bien si vous savez qu'il s'agit d'un pointeur, vous savez dès lors que déclarer une `Gtk.Window` ne fait que réservé un espace mémoire pour y stocker des adresses. Il faudra en plus absolument créer un espace mémoire sur lequel pointera notre `Gtk.Window`. Cela se faisait habituellement avec l'instruction `NEW`, mais avec GtkAda vous utiliserez la procédure `Gtk_New()` dont voici la spécification :

Code : Ada

```
procedure Gtk_New(Window : out Gtk.Window;
                  The_Type : Gtk.Enums.Gtk_Window_Type :=
```

Oublions pour l'instant le paramètre `The_Type` sur lequel nous reviendrons dans la prochaine partie. Lançons-nous ! Déclarons une `Gtk_Window` et créons sa `Gtk.Window_Record` correspondante :

Code : Ada

```
WITH Gtk.Main ;      USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;

PROCEDURE MaFenetre IS
    win : Gtk.Window ;
BEGIN
    Init ;
    gtk_new(win) ;
    Main ;
END MaFenetre ;
```



Euh... y se passe toujours rien lorsque je lance le programme ! 😱

Normal, vous avez créé une `Gtk.Window_Record`... en mémoire ! Elle existe donc bien, mais elle est tranquillement au chaud dans vos barrettes de RAM. Il va falloir spécifier que vous souhaitez l'afficher en ajoutant l'instruction `gtk.window.show()` :

Code : Ada

```
WITH Gtk.Main ;      USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;

PROCEDURE MaFenetre IS
    win : Gtk.Window ;
BEGIN
    Init ;
    Gtk.New(Win) ;
    show(win) ;
    Main ;
END MaFenetre ;
```



Une fenêtre GTK



Pourquoi la console reste-t-elle apparente lorsque je ferme la fenêtre ? 😱

Nous avons déjà en partie répondu à cette question : fermer la fenêtre revient à ne plus l'afficher, mais cela n'implique pas que vous sortez de la boucle infinie du Main. Pour en sortir, il faut associer la fonction « fermer la fenêtre » à la procédure Main.quit. Nous verrons cela plus tard, pour l'instant contentez-vous de fermer la fenêtre et la console.



La console vous permettra, lors de la phase de développement, de connaître les exceptions levées par votre programme.

Personnaliser la fenêtre



C'est bien d'avoir créé une fenêtre mais elle est petite, vide et son titre n'est pas terrible. 😞

C'est pourquoi nous allons désormais nous attacher à modifier notre fenêtre puis nous verrons dans la partie suivante comment la remplir. Je vous conseille pour mieux comprendre (ou pour aller plus loin) d'ouvrir le fichier `gtk-windows.ads` situé dans le répertoire `.../Gtakada/include/gtkada`.

Changer de type de fenêtre

Souvenez-vous, nous avons négligé un paramètre lors de la création de notre fenêtre : le paramètre `The_Type`. Sa valeur par défaut était `Gtk.Enums.window_toplevel`. Plus clairement, sa valeur était `window_toplevel` mais les valeurs de ce paramètre sont listées dans le package `Gtk.Enums` (un package énumérant diverses valeurs comme les divers types d'ancrages, les tailles d'icônes, les positions dans une fenêtre...).

Deux valeurs sont possibles :

- **Window_Toplevel** : fenêtre classique comme vous avez pu le voir tout à l'heure. Nous n'utiliserons quasiment que ce type de fenêtre.
- **Window_Popup** : fenêtre sans barre d'état ni bords, ni possibilité d'être fermée, déplacée ou redimensionnée. Vous savez, c'est le genre de fenêtre qui s'ouvre au démarrage d'un logiciel le temps que celui-ci soit chargé (Libre Office par exemple).

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ; USE Gtk.Enums ;

PROCEDURE MaFenetre IS
    win : Gtk_Window ;
BEGIN
    Init ;
    Gtk_New(win,Window_Popup) ;
    win.show ;
    Main ;
END MaFenetre ;
```

Si vous testez le code ci-dessus, vous devriez voir apparaître un magnifique rectangle gris sans intérêt à l'écran. Génial non ? 😞

Définir les paramètres avec Set_#

Définir le titre

Avant toute chose, un peu d'anglais : dans la langue de Shakespeare, le verbe « to set » (prononcez « tout cette ») a de nombreux sens tels que « fixer », « installer » ou « imposer ». Donc dès que vous aurez envie de fixer quelque chose (la taille, la position, l'apparence...), vous devrez utiliser des méthodes commençant par le terme «set». Ainsi, pour fixer le titre de votre fenêtre, nous utiliserons la méthode `set_title()` :

Code : Ada

```
BEGIN
    Init ;
    Gtk_New(win,Window_Popup) ;
    win.set_title("Super programme !") ;
    win.show ;
    Main ;
END MaFenetre ;
```

À l'inverse, si vous souhaitez connaître un paramètre, l'obtenir, il vous faudra utiliser des méthodes (plus exactement des fonctions) dont le nom commencera par le mot «get». Ainsi, `Win.Get_Title()` vous renverra le titre de la fenêtre `Win`. Exemple (regardez la console) :

Code : Ada

```
WITH Ada.Text_IO ; USE Ada.Text_IO ;
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ; USE Gtk.Enums ;

PROCEDURE MaFenetre IS
    win : Gtk_Window ;
BEGIN
    Init ;
    Gtk_New(Win,Window_Toplevel) ;
    win.set_title("Super programme !") ;
    Win.Show ;
    put_line(win.get_title) ;
    Main ;
END MaFenetre ;
```



J'insiste sur cette syntaxe : `Set` pour fixer, `Get` pour obtenir. Elle sera reprise pour toutes les méthodes de tous les éléments, que ce soient les fenêtres, les boutons, les menus, les barres de progression... Il en sera de même pour la procédure `Gtk_New`.



Si l'écriture pointée vous gêne, vous pouvez remplacer `win.set_title("Super programme !")`, `Win.Show` et `win.get_title` par `set_title(win,"super programme !"),show(win) et get_title(win)`. Mais puisque GTK est orienté objet, autant en profiter.

Fixer la taille de la fenêtre

Nous allons désormais nous focaliser sur les méthodes «set», les méthodes «get» ne nous serviront que lorsque nous pourrons interagir avec la fenêtre (et nous en sommes encore loin). Si vous avez compris ce qui a été dit juste avant, vous devriez avoir une idée du nom de la méthode permettant de fixer la taille par défaut de notre fenêtre : `Set_Default_Size`, bien sûr ! Elle prend deux paramètres : `Width` pour la largeur et `Height` pour la hauteur de la fenêtre. Exemple :

Code : Ada

```
BEGIN
    Init ;
    Gtk_New(win,Window_Popup) ;
    win.set_title("Super programme !") ;
    Win.Set_Default_Size(600,400) ;
    win.show ;
    Main ;
END MaFenetre ;
```



Si vous observez bien les spécifications du package, comme je vous l'avais conseillé, vous devriez avoir remarqué que les deux paramètres de `Set_Default_Size` ne sont pas des `Integer`, mais des `Gint` ! Il faut comprendre par là `Glib Integer`. Donc si vous voulez définir la taille par défaut à l'aide de deux variables, vous devrez soit les déclarer de type `Gint` soit de type `Integer` et les convertir en `Gint` le moment venu. Il faudra alors utiliser le package `Glib`. Vous vous souvenez ? La surcouche pour le C dont je vous parlais au premier chapitre.

Remarquez également que vous ne faites que définir une taille par défaut. Rien n'empêchera l'utilisateur de modifier cette taille à

Fenvie. Si vous souhaitez que votre fenêtre ne puisse pas être redimensionnée, vous allez devoir utiliser la méthode `set_resizable()`.

Code : Ada

```
win.set_resizable(false) ;
```

Cette ligne indiquera que votre fenêtre n'est pas «redimensionnable». En revanche, elle va automatiquement se contracter pour ne prendre que la place dont elle a besoin ; et vu que votre fenêtre ne contient rien pour l'instant, elle devient minuscule. Voici également quelques méthodes supplémentaires qui pourraient vous servir à l'avenir :

Code : Ada

```
win.fullscreen ;           --met la fenêtre en plein écran
win.unfullscreen ;         --annule le plein écran
win.resize(60,40) ;        --redimensionne la fenêtre à la
                           --taille 60x40
win.Reshow_With_Initial_Size ; --fait disparaître la fenêtre pour
                               --la réafficher
                           --à la taille initiale
```

Pour l'instant, hormis `fullscreen`, les autres méthodes ne devraient pas encore vous concerner. Mais attention, si vous mettez votre fenêtre en plein écran, vous n'aurez plus accès à l'icône pour la fermer. Vous devrez appuyer sur Alt + F4 pour la fermer ou bien sur Alt + Tab pour accéder à une autre fenêtre.

Fixer la position

Il y a deux façons de positionner votre fenêtre. La première méthode est la plus simple et la plus intuitive : `set_position()`. Cette méthode ne prend qu'un seul paramètre de type `Gtk.Enums.Gtk.Window_Position` (encore le package `Gtk.Enums`). Voici les différentes valeurs :

- `Win_Pos_None` : aucune position particulière ;
- `Win_Pos_Center` : la fenêtre est centrée à l'écran ;
- `Win_Pos_Mouse` : la fenêtre est centrée autour de la souris ;
- `Win_Pos_Center_Always` : la fenêtre est centrée à l'écran et le restera malgré les redimensionnements opérés par le programme ;
- `Win_Pos_Center_On_Parent` : la fenêtre est centrée par rapport à la fenêtre qui l'a créée.

Pour centrer la fenêtre à l'écran, il suffira d'écrire :

Code : Ada

```
WITH Gtk.Main ;          USE Gtk.Main ;
WITH Gtk.Window ;        USE Gtk.Window ;
WITH Gtk.Enums ;         USE Gtk.Enums ;

PROCEDURE MaFenetre IS
  Win : Gtk.Window ;
BEGIN
  Init ;
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Super programme !") ;
  Win.Set_Default_Size(600,400) ;
  win.set_position(win_pos_center) ;
  Win.Show ;
  Main ;
END MaFenetre ;
```

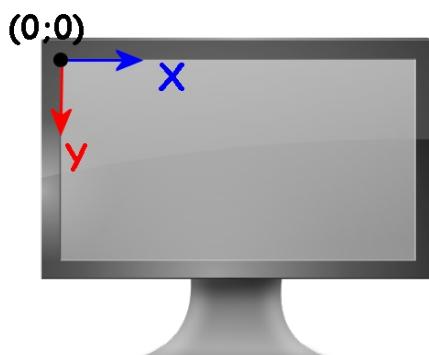
La seconde méthode pour positionner votre fenêtre est plus compliquée et consiste à la placer en comptant les pixels :

Code : Ada

```
win.move(0,0) ;
```

Cette instruction aura pour conséquence de placer votre fenêtre en haut à gauche de l'écran, ou plus exactement de placer le pixel supérieur gauche de votre fenêtre (le pixel de la fenêtre de coordonnées $(0; 0)$) dans le coin supérieur gauche de l'écran (sur le pixel de coordonnées $(0; 0)$ de votre écran). Par défaut c'est le point de la fenêtre de coordonnées $(0; 0)$ qui est le point de référence, appelé centre de gravité de la fenêtre.

Vous devez d'ailleurs savoir que chaque pixel de votre écran est repéré par des coordonnées $(x; y)$. Plus x est grand, plus le pixel est situé vers la droite. Plus y est grand, plus le pixel est situé vers le bas :



Bonus : grâce à la bibliothèque GDK et plus précisément au package `gdk.screen`, vous pouvez connaître la taille de votre écran. Les fonctions `get_width()` et `get_height()` renverront la largeur et la hauteur de votre écran. Elles prennent en paramètre des objets de type `Gdk_Screen`. Ainsi, en tapant `get_width(get_default)`, vous obtiendrez la largeur de l'écran par défaut. Ceux qui ont plusieurs écran fouilleront dans le fichier `gdk-screen.ads`.

Enfin, il est possible de faire en sorte que votre fenêtre reste au-dessus de toutes les autres, quoi que vous fassiez. Il vous suffit pour cela d'utiliser la méthode `win.set_keep_above(TRUE)`.

Fixer l'apparence

Venons-en maintenant à l'apparence de notre fenêtre. La première modification que nous pouvons apporter porte sur le bouton de fermeture de la fenêtre : actif ou non ? Par défaut, GTK crée des fenêtres supprimables, mais il est possible d'empêcher leur fermeture grâce à la méthode `win.set_deletable(FALSE)`. Pour rappel, «to delete» signifie «supprimer» en anglais, donc «deletable» signifie «supprimable».

Seconde modification possible : l'opacité de la fenêtre. Il est possible avecGtk de rendre une fenêtre complètement ou partiellement transparente :

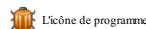


Cela se fait le plus simplement du monde avec la méthode `win.set_opacity()`. Cette méthode prend en paramètre un Gdouble compris entre 0.0 et 1.0, c'est à dire un Glib Double ce qui correspond à un nombre en virgule flottante (double précision) disponible dans le package Glib. Comme pour les Gint, vous serez sûrement amené à effectuer quelques conversions entre les float d'Ada et les Gdouble de Gdk.

Code : Ada

```
win.set_opacity(0.0) ;      --Fenêtre complètement transparente
(invisible)
win.set_opacity(0.5) ;      --Fenêtre semi-transparente comme ci-
de dessus
win.set_opacity(1.0) ;      --Fenêtre complètement opaque
```

Enfin, troisième modification (la plus attendue, je pense) : l'icône du programme. Changeons ce vilain icône de programme par un icône bien à vous. Enregistrez l'image ci-dessous dans le répertoire de votre programme (prenez soin d'avoir une image carée pour ne pas avoir de surprises à l'affichage) :



Nous allons utiliser la méthode `set.icon_from_file()` qui permet de fixer l'icône à partir d'un fichier spécifique. Cependant, contrairement aux autres méthodes utilisées jusqu'ici, il s'agit d'une fonction et non d'une procédure. Elle renvoie un booléen : si le chargement de l'icône s'est bien passé, il vaudra `TRUE`, si le fichier n'existe pas, est introuvable ou n'est pas dans un format compatible, il vaudra `FALSE`. Cela vous permet de gérer ce genre d'erreur sans faire planter votre programme ou bien de lever une exception (ce sera un excellent rappel) :

Code : Ada

```
...
LOADING_ERROR : EXCEPTION ;
BEGIN
    Init ;
    Gtk_New(Win,Window_Toplevel) ;
    Win_Set_Title("Super programme !") ;
    Win_Set_Default_Size(160,180) ;
    Win_Set_Position(Win_Pos_Center) ;
    Win_Set_Keep_Above(TRUE) ;
    Win_Set_Opacity(0.9) ;
    IF NOT Win_Set_Icon_From_File("bug-icone.png")
    THEN RAISE LOADING_ERROR ;
    END IF ;
    Win_Show ;
    Main ;
EXCEPTION
WHEN LOADING_ERROR => Put_Line("L'icone n'a pas pu être chargé") ;
END MaFenetrie ;
```



La levée de l'exception empêchera l'affichage de la fenêtre et l'exécution du Main. A vous de voir si le non chargement de l'icône mérite l'arrêt du programme.

Vous pouvez également utiliser la fonction similaire `Set_Default_Icon_From_File()`. Celle-ci ne s'applique pas à une fenêtre en particulier, elle définit un icône par défaut pour toutes les fenêtres qui n'en auraient pas. Utile pour un projet multi-fenêtre.

Ajout d'un widget

Qu'est-ce qu'un widget ?

Un Widget est l'acronyme de Windows, Icons, Dialog box, Graphics Extensions, Track ball, mais on considère plus souvent que c'est la contraction de Windows's gadget. Cela nous avance pas davantage ? Eh bien un widget, est simplement un élément graphique composant votre interface. Les boutons, les cases à cocher, les barres de défilement, les images, les lignes de texte, les zones de frappe mais aussi les menus, les icônes ou votre fenêtre elle-même constituent autant de widgets.

Pour GTK tous ces widgets constituent des classes qui dérivent en plus ou moins droite ligne de la classe `GTK_Widget_Record`. Imaginez que vous ayez un widget appelé `buzzer`, alors la norme d'écriture de GTK imposera que :

- il porte le nom de `GTK_Buzzer_Record`;
- ce soit un type `TAGGED` et `PRIVATE` dérivé de `GTK_Widget_Record`;
- il soit défini, lui et ses méthodes, dans le package `GTK.Buzzer` et que les fichiers correspondant portent les noms `gtk-buzzeraads` et `gtk-buzzeraadh`;
- un type `GTK_Buzzer` soit défini de la manière suivante : « `type GTK_Buzzer is access all GTK_Buzzer'class` »

Ces quelques remarques devraient vous rappeler quelque chose non ? C'est exactement ce que nous avons vu avec les fenêtres ! Ces conventions d'écriture et la conception orientée objet de la bibliothèque vont grandement nous simplifier la tâche, vous verrez.

Ajouter un bouton

Cessons ces généralités et ajoutons un bouton à notre fenêtre. Le widget-bouton se nomme tout simplement `GTK_Button` ! Enfin, `GTK_Button_Record` plus précisément mais vous aurez compris que c'est le pointeur `GTK_Button` qui nous intéressera réellement. Et il se trouve évidemment dans le package `Gtk.Button`. Vous commencez à comprendre le principe ? Très bien. 😊



Bouton s'écrit avec un seul T en français. En revanche, en anglais Button prend deux T.

Reprendons notre programme (que j'ai édulcoré pour ne nous concentrer que sur l'essentiel) :

Code : Ada

```
WITH Gtk.Window ;      USE Gtk.Window ;
WITH Gtk.Enums ;       USE Gtk.Enums ;
WITH Gdk.Window ;     USE Gdk.Window ;
WITH Gtk.Button ;     USE Gtk.Button ;

PROCEDURE MaFenetrie IS
    Win : Gtk.Window ;
    Bouton : Gtk.Button ;
BEGIN
    Init ;
    Gtk_New(Win,Window_Toplevel) ;
    Win_Set_Title("Super programme !") ;
    Win_Set_Default_Size(150,180) ;
    Win_Set_Position(Win_Pos_Center) ;
    IF NOT Win_Set_Icon_From_File("bug-icone.png")
    THEN NULL ;
    END IF ;
    Gtk_New(Bouton) ;
    Win.Add(Bouton) ;
    Win.Show ;
    Bouton.show ;
    Main ;
```

```
END MaFenetre ;
```

Vous remarquerez que la procédure `Gtk_New` est toujours nécessaire pour créer notre bouton. Puis nous devons l'ajouter à notre fenêtre avec l'instruction `win.add(Bouton)` sans quoi nous aurons un bouton sans trop savoir quoi en faire. Enfin, dernière étape, on affiche le bouton avec la méthode `Bouton.show`.

 Dis-moi : c'est vraiment nécessaire d'utiliser `show` autant de fois qu'il y a de widgets ? Parce que ça risque d'être un peu long quand on aura plein de bouton et de menus.

En effet, c'est pourquoi à partir de maintenant nous n'utiliserons plus `win.show` mais `win.show_all`. Cette méthode affichera non seulement la fenêtre `win` mais également tout ce qu'elle contient.

Personnaliser le bouton

 Bon c'est pas tout ça mais mon bouton est particulièrement moche. On peut pas faire mieux que ça :



Si si, la fenêtre contient un bouton

Nous pourrions commencer par lui donner un nom ou plus exactement une étiquette (`<label>` en anglais) :

Code : Ada

```
...  
Gtk_New(Bouton) ;  
Bouton.set_label("OK !") ;  
Win.add(Bouton) ;  
Win.Show_all ;
```

À noter que les lignes 20 et 21 peuvent être remplacées par une seule : `<GTK_New(Bouton, "OK !") ;>`. Bien plus rapide non ? Un autre possibilité, pour l'affichage du texte, est de souligner l'une des lettres afin d'indiquer le raccourci clavier. Mais ne vous emballez pas, nous n'allons pour l'instant souligner qu'une seule lettre. Deux méthodes s'offrent à vous :

Code : Ada

```
GTK_New(Bouton,"_Ok !") ;  
Bouton.set_use_underline(true) ; --on indique que l'on  
utilise le soulignage  
-- OU PLUS DIRECTEMENT  
GTK_New_With_Mnemonic(Bouton,"_Ok !") ; --on crée directement le  
bouton comme tel
```

L'underscore devant la lettre O indiquera qu'il faut souligner cette lettre. Si jamais vous aviez besoin d'écrire un underscore, il vous suffirait d'en écrire deux d'affilée.

 On peut placer une image dans le bouton ?

Bien sûr, mais pour cela il faudrait utiliser des `Gtk_Image` et je garde cela pour le chapitre 4. En revanche, nous pourrions changer le relief du bouton, son épaisseur. On utilise alors la méthode `Set_Relief()` qui prend comme paramètre un `Gtk.Enums.Gtk_Relief_Style` (eh oui, encore ce package `Gtk.Enums`, il liste divers styles, donc mieux vaut garder un œil dessus) :

Code : Ada

```
Bouton.Set_Relief(Relief_Normal) ; --le relief auquel vous étiez  
habitué  
Bouton.Set_Relief(Relief_Half) ; --un bouton un peu plus plat  
Bouton.Set_Relief(Relief_None) ; --un bouton complètement plat
```

Ajouter un second bouton ?

 Allez, je vais m'entraîner en ajoutant un second bouton à ma fenêtre :

Code : Ada

```
WITH Gtk.Main ; USE Gtk.Main ;  
WITH Gtk.Window ; USE Gtk.Window ;  
WITH Gtk.Enums ; USE Gtk.Enums ;  
WITH Gdk.Window ; USE Gdk.Window ;  
WITH Gtk.Button ; USE Gtk.Button ;  
  
PROCEDURE MaFenetre IS  
    Win : Gtk.Window ;  
    Btn1, Btn2 : Gtk.Button ;  
BEGIN  
    Init ;  
    Gtk_New(Win,Window_Toplevel) ;  
    Win.Set_Title("Super programme !") ;  
    Win.set_default_size(150,180) ;  
    Win.Set_Position(Win_Pos_Center) ;  
    IF NOT Win.Set_Icon_Under_Fill("bug-icon.png")  
    THEN NULL ;  
    END IF ;  
  
    Gtk_New_With_Mnemonic(Btn1, "_Ok !") ;  
    Win.Add(Btn1) ;  
    Gtk_New_With_Mnemonic(Btn2, "_Annuler !") ;  
    Win.add(Btn2) ;  
  
    Win.Show_all ;  
    Main ;  
END MaFenetre ;
```

Si jamais vous compilez ce code, vous vous apercevez que :

1. GNAT ne bronche pas. Pour lui, tout est syntaxiquement juste;
2. votre fenêtre ne comporte toujours qu'un seul bouton;
3. une belle exception semble avoir été levée car votre console vous affiche : `Gtk-WARNING**: Attempting to add a widget with type GtkButton but as a GtkBin subclass a GtkWindow can only contain one widget at a time ; it already contains a widget of type GtkButton`

Tout est dit dans la console (d'où son utilité) : les fenêtres ne peuvent contenir qu'un seul widget à la fois. Alors si vous voulez absolument placer de nouveaux boutons (car oui c'est possible), vous n'avez plus qu'à lire le chapitre 3.

Retour sur la POO

Eh bien ! Tout ce code pour seulement une fenêtre et un bouton !

C'est effectivement le problème des interfaces graphiques : elles génèrent énormément de code. Mais heureusement, nous avons déjà vu une solution idéale pour limiter tout ce code. Une solution orientée objet, adaptée aux pointeurs (et notamment aux pointeurs sur classe), qui permet d'initialiser automatiquement nos objets... Ça ne vous rappelle rien ? Les types contrôlés bien sûr ! C'est vrai que nous ne les avons pas beaucoup utilisés, même durant le TP car je savais pertinemment que nous serions amenés à les revoir.

Méthode brutale

Nous allons donc créer un package P_Fenetre, dans lequel nous déclarerons un type contrôlé T_Fenetre ainsi que sa méthode Initialize. Pas besoin de Adjust et Finalize. Notre type T_Fenetre aura deux composantes : win de type GTK_Window et btn de type GTK_Button :

Code : Ada

```
WITH Gtk.Window ;           USE Gtk.Window ;
WITH Gtk.Button ;          USE Gtk.Button ;
WITH Ada.Finalization ;    USE Ada.Finalization ;

PACKAGE P_Fenetre IS
  TYPE T_Fenetre IS NEW Controlled WITH RECORD
    Win : GTK.Window ;
    Btn : GTK.Button ;
  END RECORD ;
  PROCEDURE Initialize(F : IN OUT T_Fenetre) ;
END P_Fenetre ;
```

Que fera Initialize ? Eh bien c'est simple : tout ce que faisait notre programme précédemment. Initialize va initialiser GTK, créer et paramétriser notre fenêtre et notre bouton et lancer le Main :

Code : Ada

```
WITH Gtk.Main ;             USE Gtk.Main ;
WITH Gtk.Enums ;            USE Gtk.Enums ;
WITH Gdk.Window ;           USE Gdk.Window ;

PACKAGE BODY P_Fenetre IS
  PROCEDURE Initialize(F : IN OUT T_Fenetre) IS
  BEGIN
    Init ;                  --Création de la fenêtre
    Gtk.New(F.Win,Window_Toplevel) ;
    F.Win.Set_Title("Super programme !");
    F.Win.Default_Size(150,180) ;
    F.Win.Set_Position(Window_Pos_Center) ;
    IF NOT F.Win.Set_Icon_From_File("bug-icone.png")
      THEN NULL ;
    END IF ;
    --Création du bouton
    Gtk_New With Mnemonic(F.Btn, "_Ok !") ;
    F.Win.Add(F.Btn) ;
    --Affichage
    F.Win.Show_All ;
    Main ;
    END Initialize ;
  END P_Fenetre ;
```

Notre procédure principale devient alors extrêmement simple puisqu'il ne reste plus qu'à déclarer un objet de type T_Fenetre. Génial, non ?

Code : Ada

```
WITH P_Fenetre ;           USE P_Fenetre ;
PROCEDURE MaFenetre IS
  Mon_Programme : T_Fenetre ;
BEGIN
  NULL ;
END MaFenetre ;
```

Méthode subtile

Une seconde possibilité existe, plus nuancée : créer deux packages (P_Window et P_Button) pour deux types contrôlés (T_Window et T_Button). Cette méthode évite de créer un package fourre-tout et laisse la main au programmeur de la procédure principale. Il n'aura simplement plus à se soucier de la mise en forme. Ce qui nous donnerait les spécifications suivantes :

Code : Ada - P_Button.ads

```
WITH Gtk.Button ;           USE Gtk.Button ;
WITH Ada.Finalization ;    USE Ada.Finalization ;

PACKAGE P_Button IS
  TYPE T_Button IS NEW Controlled WITH RECORD
    Btn : GTK.Button ;
  END RECORD ;
  PROCEDURE Initialize(B : IN OUT T_Button) ;
END P_Button ;
```

Code : Ada - P_Window.ads

```
WITH Gtk.Window ;           USE Gtk.Window ;
WITH Ada.Finalization ;    USE Ada.Finalization ;
WITH P_Button ;             USE P_Button ;

PACKAGE P_Window IS
  TYPE T_Window IS NEW Controlled WITH RECORD
    Win : GTK.Window ;
  END RECORD ;
  PROCEDURE Initialize(F : IN OUT T_Window) ;
  PROCEDURE Add(W : IN OUT T_Window; B : IN T_Button) ;
  PROCEDURE Show_All(W : IN T_Window) ;
END P_Window ;
```

Et voici le corps de nos packages :

Code : Ada - P_Button.adb

```
WITH Gtk.Enums ;            USE Gtk.Enums ;
WITH Gdk.Window ;           USE Gdk.Window ;

PACKAGE BODY P_Button IS
  PROCEDURE Initialize(B : IN OUT T_Button) IS
```

```
BEGIN
  Gtk_New_With_Mnemonic(B.Btn, "_Ok !");
  B.Btn.set_relief(Relief_None);
END Initialize;
end P_Button;
```

Code : Ada - P_Window.adb

```
WITH Gtk.Enums ;           USE Gtk.Enums ;
WITH Gdk.Window ;          USE Gdk.Window ;

PACKAGE BODY P_Window IS
  PROCEDURE Initialize(F : IN OUT T_Window) IS
  BEGIN
    --Création de la fenêtre
    Gtk_New(F.Win,Window_Toplevel) ;
    F.Win.Set_Title("Super programme !");
    F.Win.set_default_size(150,180) ;
    F.Win.Set_Position(Win_Pos_Center) ;
    IF NOT F.Win.Set_Icon_From_File("bug-icon.png")
    THEN NULL ;
    END IF ;
  END Initialize ;
  PROCEDURE Add(W : IN OUT T_Window ; B : IN T_Button) IS
  BEGIN
    W.Win.Add(B.Btn) ;
  END Add ;
  PROCEDURE Show_all(W : IN T_Window) IS
  BEGIN
    W.Win.Show_all ;
  END Show_all ;
end P_Window ;
```

Et enfin notre procédure principale :

Code : Ada - MaFenetre.adb

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH P_Window ;           USE P_Window ;
WITH P_Button ;           USE P_Button ;

PROCEDURE MaFenetre IS
BEGIN
  Init ;
  DECLARE
    Win : T_Window ;
    Btn : T_Button ;
  BEGIN
    Win.Add(Btn) ;
    Win.Show_all ;
    Main ;
  END ;
END MaFenetre ;
```



Pensez à initialiser GTK avant de déclarer votre fenêtre ou votre bouton !

En résumé :

- Un programme GTK commence par `Init` et se termine par `Main`.
- Un widget GTK commence par «`GTK_`» suivi de son nom. Le package correspondant commence «`Gtk`» suivi lui aussi du nom du widget.
- Avant de manipuler les widgets, pensez à les initialiser avec `Gtk_New()`. Avant le `Main`, pensez à les afficher avec les méthodes `show` ou `show_all`.
- Pour définir les paramètres d'un widget, on utilise les méthodes dont le nom commence par «`set_`» suivi de l'élément à paramétrier.
- Pour clarifier le code de la procédure principale, pensez à créer des types contrôlés.

Les conteneurs I

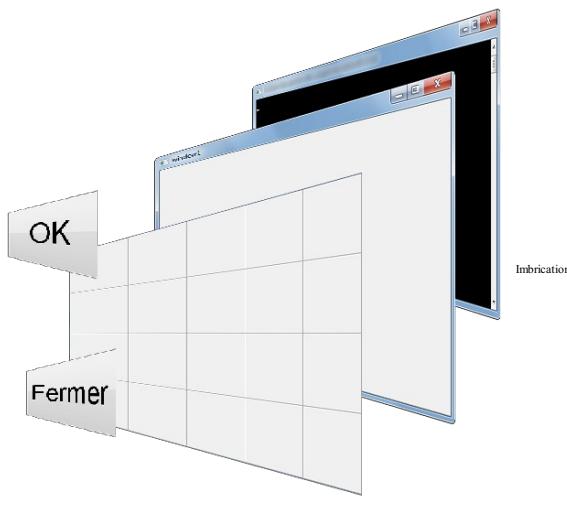
Nous nous sommes arrêtés au dernier chapitre sur une déception de taille : notre fenêtre ne comporte qu'un seul gros bouton qui occupe la totalité de la place et qui est redimensionné en même temps que la fenêtre. De plus, il est impossible d'ajouter un second bouton à notre fenêtre. Alors pour avoir plusieurs boutons dans votre application, il y a une solution : les conteneurs. Ces widgets un peu particuliers font l'objet d'un chapitre à part entière.

Des conteneurs pour... contenir !

Qu'est-ce qu'un conteneur ?

Non, quand je parle de conteneur je ne parle pas de des grosses caisses métalliques voyageant sur les camions ou les bateaux !!! s'agit de widgets très particuliers, pas nécessairement visibles mais dont le seul rôle est de contenir d'autres widgets. Les fenêtres ou les boutons dérivent de la classe `GTK.Container_Record` puisqu'une fenêtre peut contenir un bouton et qu'un bouton peut contenir une étiquette (du texte) et une image. Mais, quand je parlerai de conteneurs, je sous-entendrai désormais des conteneurs conçus pour organiser nos fenêtres.

L'idée est simple : une fenêtre ne peut contenir qu'un seul widget. Mais il est possible de créer des widgets appelés conteneurs pouvant contenir plusieurs autres widgets. Alors pourquoi ne pas placer tous nos widgets dans un seul conteneur que l'on placera quant à lui dans la fenêtre ? Nous obtiendrons quelque chose comme cela :



des widgets et des conteneurs

Présentation de différents conteneurs

Je vais vous présenter succinctement les quelques conteneurs que nous verrons dans ce chapitre. Il en existe d'autres que nous verrons plus tard, nous allons pour l'instant nous concentrer sur les conteneurs principaux.

Les alignements

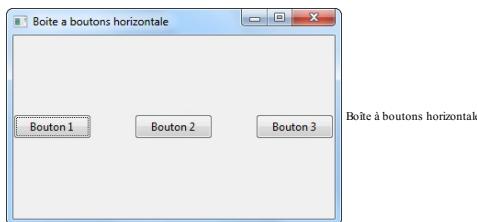
Il s'agit d'un conteneur prévu pour un seul sous-widget (ce dernier est appelé **widget enfant**). Mais il permet, comme son nom l'indique, d'aligner un bouton selon des critères précis, par exemple : *«je veux que le bouton reste en bas à gauche de la fenêtre»*. Cela évite d'avoir un gros bouton tout moche qui se déforme en même temps que la fenêtre. Je sais, c'est pas folichon, mais il est essentiel de comprendre son fonctionnement avant de s'attaquer à des conteneurs plus complexes.



Conteneur d'alignement

Les boîtes

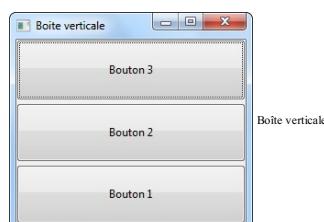
Les boîtes peuvent quant à elles avoir plusieurs widgets alignés verticalement ou horizontalement.



Boîte à boutons horizontale



Boîte horizontale



Boîte verticale

Les tables

Les tables ressemblent peu ou prou aux boîtes mais proposent d'afficher plusieurs lignes ou colonnes de widgets.



Table de boutons

Les conteneurs à position fixée

Ces conteneurs permettent de fixer un widget selon des coordonnées. Plus simple d'emploi, il est en revanche plus compliqué d'arriver à des interfaces agréables, sans bugs et portables aisément sur d'autres ordinateurs avec ces conteneurs.



Boutons dans un conteneur à position fixe

Les alignements

Fiche d'identité

Nous allons voir désormais de nombreux widgets, alors afin de ne pas me répéter, je vous présenterai désormais ces nouveautés par une rapide fiche d'identité.

- **Widget :** GTK_Alignment
- **Package :** Gtk.Alignment
- **Descendance :** GTK_Widget >> GTK_Container >> GTK_Bin (conteneurs pour un unique widget)
- **Description :** Conteneur permettant d'aligner de façon fine un unique widget dans une fenêtre.

Créer un GTK_Alignment

Repartons sur un code relativement épuré pour ne pas nous méler les pinceaux :

Code : Ada

```
WITH Gtk.Main ;      USE Gtk.Main ;
WITH Gtk.Window ;    USE Gtk.Window ;
WITH Gtk.Enums ;    USE Gtk.Enums ;
WITH Gtk.Button ;   USE Gtk.Button ;

PROCEDURE MaFenetre IS
  Win    : Gtk.Window ;
  Btn    : Gtk_Button ;
BEGIN
  Init ;
  Gtk_New(Win,Winow_Toplevel) ;
  Win.Set_Title("Alignment") ;
  Win.set_default_size(250,200) ;
  Gtk_New(Btn, "Bouton") ;
  Win.add(Btn) ;
  Win.Show_all ;
  Main ;
END MaFenetre ;
```

Si, comme je vous l'avais conseillé, vous avez ouvert les spécifications du package `Gtk.Alignment`, vous pourrez vous rendre compte que la procédure `Gtk_New()` a subi quelques modifications pour les `GTK_Alignment` :

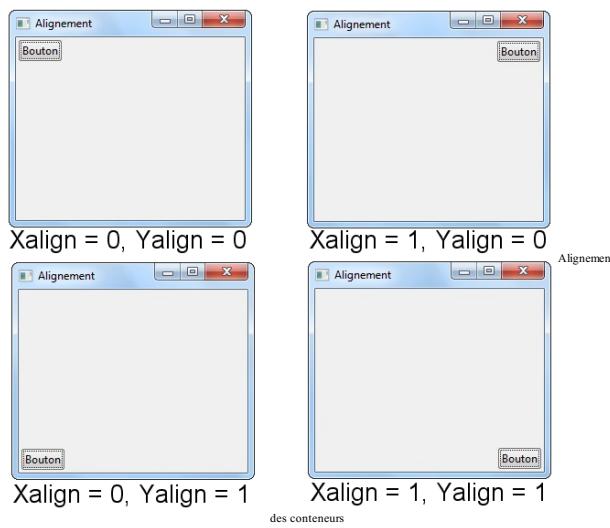
Code : Ada

```
procedure Gtk_New
  (Alignment : out Gtk_Alignment;
   Xalign   : Gfloat;
   Yalign   : Gfloat;
   Xscale   : Gfloat;
   Yscale   : Gfloat);
```

Je passe sur le type `GFloat`, chacun aura compris qu'il s'agit d'un type virgule flottante défini très certainement par Cib. La question est surtout : que sont tous ces paramètres ?

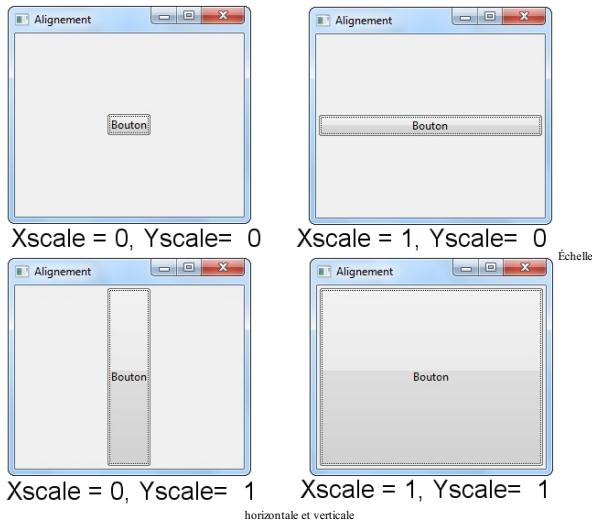
- Le paramètre `Xalign` permet de positionner votre conteneur horizontalement : si `Xalign = 0.0` alors le widget est complètement à gauche. Si `Xalign = 1.0`, le conteneur sera le plus à droite possible. Si `Xalign = 0.5`, le conteneur sera situé au milieu (horizontalement).
- Le paramètre `Yalign` permet de positionner votre conteneur verticalement. Si `Yalign = 0.0`, votre conteneur sera tout en haut ; si `Yalign = 1.0`, votre conteneur sera tout en bas.

Ces deux premiers paramètres ont des valeurs entre `0.0` et `1.0` et doivent être vus comme des pourcentages : `Xalign = 0.50` et `Yalign = 1.00`, signifie que le conteneur est aligné à 50% selon l'horizontale et à 100% selon la verticale. Autrement dit, il se trouve au milieu en bas.



- Le paramètre `Xscale` signifie littéralement «échelle horizontale». Il correspond à «l'étalement» du conteneur horizontalement. Les images ci-dessus ont été faites avec un `Xscale` valant `0.0`, c'est-à-dire que le conteneur ne s'étale pas, il prend le moins de place possible, la seule limite étant la place nécessaire à l'affichage du texte du bouton. Si `Xscale = 1.0`(sa valeur maximale), alors le conteneur prendra 100% de la place disponible horizontalement.
- Le paramètre `Yscale` a lui le même rôle mais verticalement, vous l'aurez compris.

Mieux vaut une petite image pour bien comprendre. Je vais placer mon conteneur au centre (`Xalign = 0.50` et `Yalign = 0.50`) et faire varier `Xscale` et `Yscale`:



Sachant cela, nous allons créer un Alignement contenant un bouton, situé en haut au milieu et s'étalant de 10% horizontalement et 0% verticalement. Il faudra penser à ajouter l'alignement à la fenêtre puis à ajouter le bouton à l'alignement (ou l'inverse):

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ;          USE Gtk.Enums ;
WITH Gtk.Button ;         USE Gtk.Button ;
WITH Gtk.Alignment ; USE Gtk.Alignment ;

PROCEDURE MaFenetre IS
  Win      : Gtk.Window ;
  Conteneur : Gtk.Alignment ;
  Btn      : Gtk_Button ;
BEGIN
  Init ;
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Alignment") ;
  Win.Set_Default_Size(250,200) ;
  Gtk_New(Conteneur,0.5,0.0,0.1,0.0) ;
  Win.Add(Conteneur) ;
  Gtk_New(Btn, "Bouton") ;
  Conteneur.Add(Btn) ;
  Win.Show_All ;
  Main ;
END MaFenetre ;
```

Ces paramètres pourront être modifiés plus tard grâce à la méthode `conteneur.set()` :

Code : Ada

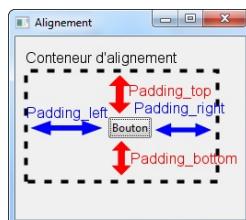
```
procedure Set
  (Alignment : access Gtk_Alignment_Record;
   Xalign   : Gfloat;
   Yalign   : Gfloat;
   Xscale   : Gfloat;
   Yscale   : Gfloat);
```

Le padding

Le padding correspond à l'écart séparant le bord du conteneur du bord du bouton. Pour l'instant, le padding est de 0 pixels, ce qui veut dire que le conteneur enserre parfaitement le bouton. Pour créer un peu d'air (et éviter à terme que d'autres widgets viennent se coller à votre conteneur) nous pouvons utiliser cette méthode :

Code : Ada

```
procedure Set_Padding
  (Alignment : access Gtk_Alignment_Record;
   Padding_Top : Uint; --espace au dessus du bouton
   Padding_Bottom : Uint; --espace en dessous du bouton
   Padding_Left : Uint; --espace à gauche du bouton
   Padding_Right : Uint); --espace à droite du bouton
```



Je vous conseille d'avoir un padding de seulement quelques pixels (3 par exemple) : «conteneur.set_padding(3,3,3,3)».

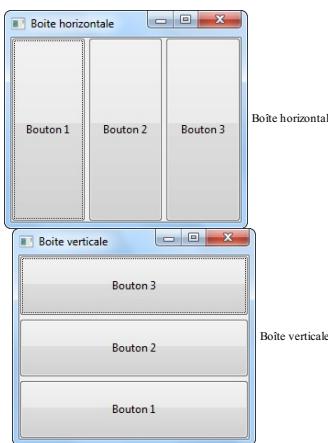
Les boîtes Boîtes classiques

Fiche d'identité

- **Widgets :** GTK_Box, GTK_HBox, GTK_VBox
- **Package :** Gtk.Box
- **Descendance :** GTK_Widget >> GTK_Container
- **Description :** Conteneurs permettant d'aligner plusieurs widgets horizontalement (HBox) ou verticalement (VBox).

Créer des boîtes

Si vous regardez en détail le début des spécifications du package, vous remarquerez qu'il y a trois widgets : GTK_Box et deux sous-types GTK_HBox et GTK_VBox. Ce sont ces deux derniers que nous allons utiliser : GTK_HBox est une boîte horizontale et GTK_VBox est une boîte verticale. Il existe donc un constructeur pour chacun de ces widgets : GTK_New_HBox() et GTK_New_VBox().



Code : Ada

```
procedure Gtk_New_Hbox
  (Box : out Gtk_Hbox;
   Homogeneous : Boolean := False;
   Spacing : Gint := 0);
procedure Gtk_New_Vbox
  (Box : out Gtk_Vbox;
   Homogeneous : Boolean := False;
   Spacing : Gint := 0);
```

Si le paramètre Homogeneous vaut **TRUE**, alors les widgets auront des tailles similaires, homogènes. Le paramètre Spacing est similaire au paramètre Padding des alignements : il indique le nombre de pixels (l'espace) qui séparera les widgets enfants. Ces deux paramètres pourront être modifiés plus tard grâce aux méthodes **Set_Homogeneous()** et **Set_Spacing()**. Créons par exemple une boîte verticale homogène avec un espacement de 3 pixels.

Code : Ada

```
WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Enums ; USE Gtk.Enums ;
WITH Gtk.Button ; USE Gtk.Button ;
WITH Gtk.Box ; USE Gtk.Box ;

PROCEDURE MaFenetre IS
  Win : Gtk_Window ;
  Boite : Gtk_VBox ;
  Btn1, Btn2, Btn3 : Gtk_Button ;
BEGIN
  Init ;
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Boîte verticale") ;
  win.set_default_size(250,200) ;

  Gtk_New(Boite, homogenous => true,
  spacing => 3) ;
  Win.Add(Boite) ;

  Gtk_New(Btn1, "Bouton 1") ;
  Gtk_New(Btn2, "Bouton 2") ;
  Gtk_New(Btn3, "Bouton 3") ;
  -- ici, nous ajouterons les trois boutons dans notre boîte
  Win.Show_all ;
  Main ;
END MaFenetre ;
```

Ajout de widgets

Reste maintenant à intégrer nos trois boutons à notre boîte verticale. La méthode **add()** est inopérante avec les boîtes : ajouter des widgets, c'est bien gentil, mais où et comment les ajouter ? Pour cela, imaginez une grande boîte en carton posée devant vous. Vous posez un objet au centre puis, pour gagner de la place, vous le poussez soit vers le haut du carton, soit vers le bas.

C'est exactement ce que nous allons faire avec nos widgets en les «poussant autant que possible» soit vers le début (le haut pour les Vbox, la gauche pour les HBox), soit vers la fin (le bas ou la droite). Pour «pousser» un widget vers le haut de notre GTK_VBox, vous utiliserez la méthode `Pack_start()`. Pour «pousser» un widget vers le bas, vous utiliserez la méthode `Pack_end()`.

Code : Ada

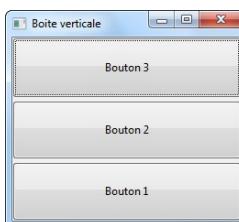
```
WITH Gtk.Main ;      USE Gtk.Main ;
WITH Gtk.Window ;   USE Gtk.Window ;
WITH Gtk.Enums ;    USE Gtk.Enums ;
WITH Gtk.Button ;   USE Gtk.Button ;
WITH Gtk.Box ;       USE Gtk.Box ;

PROCEDURE MaFenetre IS
  Win          : Gtk.Window ;
  Boite        : Gtk_VBox ;
  Btn1, Btn2, Btn3 : Gtk_Button ;
BEGIN
  Init ;
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Boîte verticale") ;
  Win.set_default_size(250,200) ;

  Gtk_New_VBox(Boite,
               homogeneous => true,
               spacing => 3) ;
  Win.Add(Boite) ;

  Gtk_New(Btn1, "Bouton 1") ; Boite.Pack_End(Btn1) ;
  Gtk_New(Btn2, "Bouton 2") ; Boite.Pack_End(Btn2) ;
  Gtk_New(Btn3, "Bouton 3") ; Boite.Pack_End(Btn3) ;

  Win.Show_all ;
  Main ;
END MaFenetre ;
```



Assemblage avec Pack_end()



Si l'ordre obtenu vous déplaît, vous pouvez repositionner certains widgets. Exemple : Boite.reorder_child(Btn1,2) repositionnerait le Bouton 1 en deuxième position.

Si vous testez le code ci-dessus, vous remarquerez que Bn1 est le premier bouton placé à la fin : ce sera donc le bouton le plus en bas.

Paramétrez vos widgets enfants

Maintenant, si vous avez regardé les spécifications de ces méthodes, vous aurez remarqué qu'elles comportent trois autres paramètres : `Expand`, `Fill` et `Padding`.

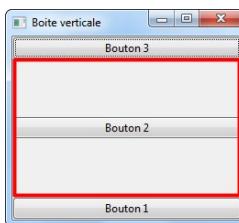
- Vous savez désormais ce qu'est le padding, les méthodes `Pack_End()` et `Pack_Start()` vous proposent un padding supplémentaire pour certains widget, `Padding` qui s'ajoute au Spacing de la boîte.
- Le paramètre `Expand`, s'il vaut `TRUE`, indique que l'encart contenant votre widget va pouvoir «prendre ses aises» et s'étendre autant qu'il le peut, même au détriment de ses camarades. Si la fenêtre est redimensionnée, l'encart le sera aussi. Ce paramètre n'a aucun intérêt si vous avez paramètre `Homogeneous` à `TRUE`.
- Le paramètre `Fill`, s'il vaut `TRUE`, indique que le widget doit occuper toute l'espace disponible dans son encart. S'il vaut `FALSE`, le widget se limitera à l'espace dont il a réellement besoin. Ce paramètre n'a d'intérêt que si `Expand` vaut `TRUE`.

Pour mieux comprendre, reprenons notre exemple :

Code : Ada

```
...Gtk_New_VBox(Boite,
               homogeneous => false,
               spacing => 3) ;

  Gtk_New(Btn1, "Bouton 1") ; Boite.Pack_End(Btn1, Expand => false,
                                               Fill => false) ;
  Gtk_New(Btn2, "Bouton 2") ; Boite.Pack_End(Btn2, Expand => true,
                                               Fill => true) ;
  Gtk_New(Btn3, "Bouton 3") ; Boite.Pack_End(Btn3, Expand => false,
                                               Fill => false) ;
```

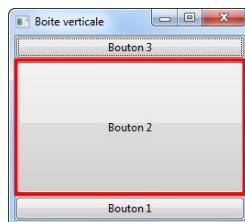


Expand vaut TRUE pour le bouton 2

Pour le bouton2, `Expand = true`, donc son encart prend toute la place disponible, au détriment des autres.

Code : Ada

```
...Gtk_New(Btn1, "Bouton 1") ; Boite.Pack_End(Btn1, Expand => false,
                                               Fill => false) ;
  Gtk_New(Btn2, "Bouton 2") ; Boite.Pack_End(Btn2, Expand => true,
                                               Fill => true) ;
  Gtk_New(Btn3, "Bouton 3") ; Boite.Pack_End(Btn3, Expand => false,
                                               Fill => false) ;
```



Expand et Fill sont vrais pour le bouton 2

Cette fois, Expand = **TRUE** et Fill = **TRUE**, donc son encart prend toute la place disponible et le bouton remplit l'encart.

Avec plusieurs types de widgets

Un avantage des boîtes, c'est qu'elles peuvent contenir divers widgets, pas nécessairement de même type. Par exemple, nous allons remplacer le troisième bouton par un GTK_Label, c'est-à-dire une étiquette (du texte quoi) :

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ;          USE Gtk.Enums ;
WITH Gtk.Button ;         USE Gtk.Button ;
WITH Gtk.Box ;             USE Gtk.Box ;
WITH Gtk.Label ;           USE Gtk.Label ;

PROCEDURE MaFenetre IS
  Win          : Gtk.Window ;
  Boite        : Gtk_VBox ;
  Btn1, Btn2  : Gtk_Button ;
  Lbl          : GTK_Label ;
BEGIN
  Init ;
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Boîte verticale") ;
  win.set_default_size(250,200) ;

  Gtk_New_VBox(Boite,
               homogeneous => false,
               spacing => 3) ;
  Win.Add(Boite) ;

  Gtk_New(Btn1, "Bouton 1") ; Boite.Pack_End(Btn1, Expand => false,
                                               Fill => false) ;
  Gtk_New(Btn2, "Bouton 2") ; Boite.Pack_End(Btn2, Expand => true,
                                               Fill => true) ;
  Gtk_New(Lbl, "Ceci est du texte") ; Boite.Pack_End(Lbl, Expand =>
False, Fill => false) ;

  Win.Show_all ;
  Main ;
END MaFenetre ;
```



Boîte contenant deux types de widget

Avoir tous ces différents widgets ne posera pas de soucis à GTK ni à notre boîte.



Nous reverrons les GTK_Label plus en détail, rassurez-vous ☺

Mélanger le tout



Ça veut dire que mes widgets seront soit à l'horizontale, soit à la verticale, mais qu'il n'y a pas moyen de panacher...
c'est un peu limité quand même ☹

Allons, soyez imaginatifs. Les conteneurs sont des widgets faits pour contenir d'autres widgets. Il n'est donc pas interdit qu'un conteneur contienne un autre conteneur ! Par exemple, notre GTK_VBox, pourrait contenir une GTK_HBox à l'emplacement du bouton 2 !

Code : Ada

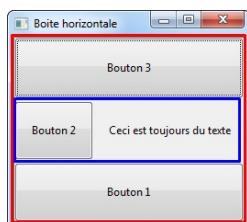
```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ;          USE Gtk.Enums ;
WITH Gtk.Button ;         USE Gtk.Button ;
WITH GTK.Label ;           USE Gtk.Label ;
WITH Gtk.Box ;             USE Gtk.Box ;

PROCEDURE MaFenetre IS
  Win          : Gtk.Window ;
  VBoite       : Gtk_VBox ;
  HBoite       : Gtk_HBox ;
  Btn1, Btn2, Btn3 : Gtk_Button ;
  Lbl          : GTK_Label ;
BEGIN
  Init ;
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Boîte horizontale") ;
  win.set_default_size(250,200) ;

  Gtk_New_VBox(VBoite,
               homogeneous => false,
               spacing => 3) ;
  Win.Add(VBoite) ;

  --On remplit la boîte verticale
  Gtk_New(Btn1, "Bouton 1") ;
  Gtk_New_HBox(HBoite, homogeneous => false, Spacing => 3) ;
  Gtk_New(Btn3, "Bouton 3") ;
  VBoite.Pack_End(Btn1) ;
  VBoite.Pack_End(HBoite) ;
  VBoite.Pack_End(Btn3) ;
  --On remplit la boîte horizontale
  Gtk_New(Btn2, "Bouton 2") ;
  Gtk_New(Lbl, "Ceci est toujours du texte") ;
  HBoite.Pack_Start(Btn2) ;
  HBoite.Pack_Start(Lbl) ;

  Win.Show_all ;
  Main ;
END MaFenetre ;
```



VBox contenant une HBox

Comme vous pouvez le constater, la GTK_VBox comprend deux boutons ainsi qu'une GTK_HBox, laquelle contient une étiquette et un bouton.

Boîtes à boutons

Fiche d'identité

- **Widgets** : GTK_HButton_Box et GTK_VButton_Box
- **Package** : GTK_Button_Box GTK_HButton_Box et GTK_VButton_Box
- **Descendance** : GTK_Widget >> GTK Container >> GTK_Box >> GTK_Button_Box
- **Description** : Conteneurs spécifiquement prévu pour organiser les boutons de façon harmonieuse, horizontalement ou verticalement.

Créer et remplir une boîte à boutons

Nos boîtes, si elles sont très pratiques, ne sont malheureusement pas très esthétiques. Nos boutons sont soit écrasés soit énormes. Et cela ne s'arrange pas si l'on agrandit la fenêtre. Pour pallier à ce problème existe un deuxième type de boîte : les boîtes à boutons. Comme pour les boîtes classiques, il existe un modèle vertical et un modèle horizontal. Toutefois, chacun a son package spécifique. Celles-ci dérivant du type GTK_Button_Box, lui-même dérivant du type GTK_Box, les méthodes pack_end() et pack_start() vues précédemment restent valides :

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ;          USE Gtk.Enums ;
WITH Gtk.Button ;          USE Gtk.Button ;
WITH Gtk.HButton_Box ; USE Gtk.HButton_Box ;

PROCEDURE MaFenetre IS
  Win : Gtk.Window ;
  Boite : Gtk.HButton_Box ;
  Btn1, Btn2, Btn3 : Gtk.Button ;
BEGIN
  Init ;
  Gtk_New(Win, Window_Toplevel) ;
  Win.Set_Title("Boîte à boutons horizontale") ;
  Win.set_default_size(350, 200) ;

  Gtk_New(Boite) ;
  Win.Add(Boite) ;

  Gtk_New(Btn1, "Bouton 1") ; Boite.Pack_End(Btn1) ;
  Gtk_New(Btn2, "Bouton 2") ; Boite.Pack_End(Btn2) ;
  Gtk_New(Btn3, "Bouton 3") ; Boite.Pack_End(Btn3) ;

  Win.Show_all ;
  Main ;
END MaFenetre ;
```



Ainsi, les boutons sont positionnés de manière agréable sans avoir à se soucier des Padding et autres Spacing. Malgré l'agrandissement de la fenêtre, les boutons garderont un aspect normal.

Paramétrages spécifiques à la boîte à boutons

Les boîtes à boutons présentent de nouveaux paramètres pour affiner la présentation. Tout d'abord avec la méthode Set_Layout() (package GTK_Button_Box) ou la méthode Set_Layout_Default() (packages GTK_HButton_Box et GTK_VButton_Box). Celes-ci proposent différents placements par défaut pour vos boutons. Ces placements sont de type Gtk.Enums.Gtk_Button_Box_Style (encore Gtk.Enums !) dont voici les valeurs possibles :

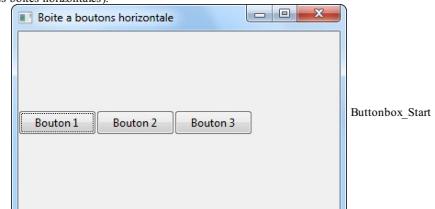
- **Buttonbox_Spread** : les boutons sont placés de manière régulière dans la boîte.



- **Buttonbox_Edge** : les boutons sont placés de manière régulière, mais le premier et le dernier boutons sont collés aux bords de la boîte.



- **Buttonbox_Start** : les boutons sont placés le plus près possible du début de la boîte (le haut pour les boîtes verticales, la gauche pour les boîtes horizontales).



- **Buttonbox_End** : les boutons sont placés le plus près possible de la fin de la boîte (le bas pour les boîtes verticales, la droite pour les boîtes horizontales).



Autre personnalisation possible : rendre secondaire certains boutons. Cela est souvent utilisé pour les boutons «Aide» ou «Options». Ces derniers sont mis à l'écart des autres. Les GTK_Button_Box permettent cela grâce à la méthode `Set_Child_Secondary()`. Par exemple, si nous modifions notre code pour que les boutons soient alignés sur la gauche et que le bouton n°3 soit secondaire, cela nous donnera :

Code : Ada

```
...
Gtk_New(Boite) ;
Boite.set_layout(Buttonbox_Start) ;
Win.Add(Boite) ;

Gtk_New(Btn1, "Bouton 1") ; Boite.Pack_Start(Btn1) ;
Gtk_New(Btn2, "Bouton 2") ; Boite.Pack_Start(Btn2) ;
Gtk_New(Btn3, "Bouton 3") ; Boite.Pack_Start(Btn3) ;
Boite.set_child_secondary(child => Btn3, Is_secondary => true) ;
```



Le bouton 3 est secondaire

[Les tables](#)

[Fiche d'identité](#)

- **Widget** : GTK_Table
- **Package** : Gtk.Table
- **Descendance** : GTK_Widget >> GTK_Container
- **Description** : Conteneur permettant de stocker plusieurs widgets répartis sur plusieurs lignes et plusieurs colonnes.



Table à widgets

[Créer une table de widgets](#)

Vous devriez rapidement comprendre le fonctionnement du constructeur à la lecture de sa spécification :

Code : Ada

```
procedure Gtk_New
  (Table      : out Gtk_Table;
   Rows       : Guint;
   Columns   : Guint;
   Homogeneous : Boolean);
```

En créant votre table, vous devrez indiquer le nombre de lignes (rows), de colonnes (columns) et indiquer si la table sera homogène (reliez la partie sur les boîtes si vous avez déjà oublier ce que cela signifie). Pour information, les types Guint sont encore des types entiers comme les Gint et signifient «Glib Unsigned Integer», autrement dit, des entiers positifs. En cas d'erreur, vous pourrez toujours revenir ces paramètres à l'aide des deux méthodes suivantes :

Code : Ada

```
procedure Resize
  (Table : access Gtk_Table_Record;
   Rows  : Guint;
   Columns : Guint);
procedure Set_Homogeneous
  (Table : access Gtk_Table_Record;
   Homogeneous : Boolean);
```

Vous pouvez également régler l'écart entre les colonnes ou entre les lignes avec les deux méthodes ci-dessous. L'écart correspond bien entendu au nombre de pixels.

Code : Ada

```
procedure Set_Col_Spacings          --définit      l'espace
entre toutes les colonnes
  (Table : access Gtk_Table_Record;
   Spacing : Guint);
procedure Set_Col_Spacing           --définit l'espace entre
deux colonnes spécifiques
  (Table : access Gtk_Table_Record;
   Column : Guint;
   Spacing : Guint);

procedure Set_Row_Spacings          --définit      l'espace
entre toutes les lignes
  (Table : access Gtk_Table_Record;
   Spacing : Guint);
procedure Set_Row_Spacing           --définit l'espace entre
deux lignes spécifiques
  (Table : access Gtk_Table_Record;
   Row : Guint;
   Spacing : Guint);
```

Voici le code initial de notre fenêtre :

Code : Ada

```

WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;        USE Gtk.Window ;
WITH Gtk.Enums ;         USE Gtk.Enums ;
WITH Gtk.Button ;        USE Gtk.Button ;
WITH Gtk.Table ; USE Gtk.Table ;

PROCEDURE MaFenetre IS
  Win          : Gtk.Window ;
  Tableau : Gtk.Table ;
  Btn1, Btn2, Btn3   : Gtk.Button ;
  Btn4, Btn5, Btn6   : Gtk.Button ;
BEGIN
  Init ;
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Table a widgets") ;
  Win.set_default_size(350,200) ;
  Tableau := Gtk_New(Tableau,3,2,TRUE) ;
  Tableau.Set_Row_Spacings(1) ;
  Tableau.Set_Col_Spacings(2) ;
  Win.Add(Tableau) ;
  --Nous ajouterons ici les boutons à notre table
  Win.Show_all ;
  Main ;
END MaFenetre ;

```

Ajouter des widgets

Avant de commencer à ajouter des boutons à notre table, vous devez savoir qu'un widget peut occuper plusieurs cases alors que son voisin n'en occupe qu'une seule. Cette liberté offerte au programmeur va quelque peu alourdir notre code car nous devrons renseigner où se situe les bords gauche, droite, haut et bas du widget. Pour comprendre de quoi je parle, regardez la spécification (allégée) de la méthode pour ajouter des widgets :

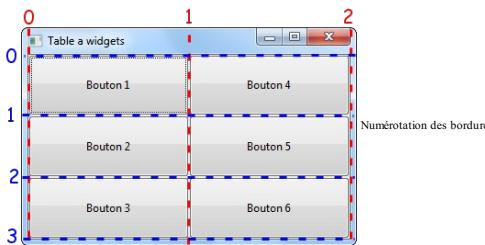
Code : Ada

```

procedure Attach      --ou Attach_Default
  (Table : access Gtk_Table_Record;
   Widget : access Gtk.Widget.Gtk_Widget_Record'Class;
   Left_Attach : Guint;
   Right_Attach : Guint;
   Top_Attach : Guint;
   Bottom_Attach : Guint);

```

Vous devez vous imaginer que chaque bordure ou séparation de la table porte un numéro. La première bordure, celle du haut ou du gauche, porte le numéro 0. La bordure de droite porte le numéro du nombre de colonnes ; la bordure du bas porte le numéro du nombre de lignes :



Ainsi, on peut affirmer que le bouton n°2 s'étend de 0 à 1 horizontalement et de 1 à 2 verticalement, ce qui se traduit en langage Ada par :

Code : Ada

```

Tableau.Attach(
  Widget    => Btn2 ;
  Left_Attach  => 0 ;
  Right_Attach => 1 ;
  Top_Attach   => 1 ;
  Bottom_Attach => 2);

```

Pour obtenir la table de widgets donnée en exemple, voici donc ce que j'ai écrit :

Code : Ada

```

...
Gtk_New(Btn1, "Bouton 1") ; Tableau.attach(Btn1,0,1,0,1) ;
Gtk_New(Btn2, "Bouton 2") ; Tableau.attach(Btn2,1,1,1,1) ;
Gtk_New(Btn3, "Bouton 3") ; Tableau.attach(Btn3,1,1,2,3) ;
Gtk_New(Btn4, "Bouton 4") ; Tableau.attach(Btn4,1,2,1,1) ;
Gtk_New(Btn5, "Bouton 5") ; Tableau.attach(Btn5,1,2,2,3) ;
Gtk_New(Btn6, "Bouton 6") ; Tableau.attach(Btn6,2,2,2,3) ;

```

Les paramètres supplémentaires

Malheureusement, je vous ai bien dit que le constructeur que je vous avais transmis était allégé. Le véritable constructeur a plutôt la spécification suivante :

Code : Ada

```

procedure Attach
  (Table      : access Gtk_Table_Record;
   Child     : access Gtk.Widget.Gtk_Widget_Record'Class;
   Left_Attach : Guint;
   Right_Attach : Guint;
   Top_Attach : Guint;
   Bottom_Attach : Guint;
   Koptions : Gtk.Enums.Gtk_Table_Options := Expand or Fill;
   Yoptions : Gtk.Enums.Gtk_Attach_Options := Expand or Fill;
   Xpadding : Guint := 0;
   Ypadding : Guint := 0);

```

Vous vous souvenez du padding ? Nous l'avions vu avec les alignements. Il s'agit du nombre de pixels séparant le bord de l'enfant et le widget qui s'y trouve. Xpadding indique donc l'écart horizontal, Ypadding l'écart vertical. Attention, le padding n'influe pas sur le conteneur (sauf si la fenêtre est trop petite) mais sur la taille du widget enfant.

Les paramètres Xoptions et Yoptions correspondent au comportement du widget à l'horizontale et à la verticale : notre bouton doit-il prendre le maximum ou le minimum de place ? Les valeurs possibles sont les suivantes (disponibles avec Gtk.Enums, encore et toujours) :

- **Expand** : déjà vu avec les boîtes. Le widget tente de maximiser la place occupée par son encart. N'a vraiment de sens que pour une table non homogène.
- **Fill** : déjà vu avec les boîtes. Le widget remplit entièrement l'enfant qui lui est réservé.
- **Expand or Fill** : permet de combiner les deux caractéristiques précédentes. D'ailleurs, c'est la valeur par défaut du paramètre.
- **Shrink** : indique que le widget ne prendra que la place strictement nécessaire.

Dans les faits, soit vous ne modifierez pas ces paramètres et votre bouton occupera tout l'espace alloué, soit vous utiliserez la

valeur shrink pour restreindre la taille du widget au maximum. Modifions notre code pour y voir plus clair et supprimons l'un des boutons :

Code : Ada

```
WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Enums ; USE Gtk.Enums ;
WITH Gtk.Button ; USE Gtk.Button ;
WITH Gtk.Table ; USE Gtk.Table ;

PROCEDURE MaFenetre IS
    Win : Gtk.Window ;
    Tableau : Gtk.Table ;
    Btn1, Btn2, Btn3 : Gtk.Button ;
    Btn4, Btn5 : Gtk.Button ;
BEGIN
    Init ;
    Gtk_New(Win, Window_Toplevel) ;
    Win.Set_Title("Table a widgets") ;
    Win.set_default_size(350, 200) ;

    Gtk_New(Tableau, 3, 2, false) ;
    Tableau.Set_Row_Spacings(1) ;
    Tableau.Set_Col_Spacings(2) ;
    Win.Add(Tableau) ;

    Gtk_New(Btn1, "Bouton 1") ; Tableau.attach(Btn1, 0, 1, 0, 1, Xoptions =>
shrink, Yoptions => fill) ;
    Gtk_New(Btn2, "Bouton 2") ; Tableau.attach(Btn2, 0, 1, 1, 2, Xoptions =>
shrink, Yoptions => shrink) ;
    Gtk_New(Btn3, "Bouton 3") ; Tableau.attach(Btn3, 0, 1, 2, 3, Xoptions =>
fill, Yoptions => fill) ;
    Gtk_New(Btn4, "Bouton 4") ; Tableau.Attach(Btn4, 1, 2, 0, 1, Xpadding =>
25, Xoptions => fill,
Yoptions => shrink) ;
    Gtk_New(Btn5, "Bouton 5") ; Tableau.Attach(Btn5, 1, 2, 1, 3, Xoptions =>
expand or fill,
Yoptions => expand or fill) ;

    Win.Show_all ;
    Main ;
END MaFenetre ;
```

Je vous laisse comparer les effets des différents paramètres sur les boutons de ma nouvelle fenêtre :



Le widget pour position fixe

Fiche d'identité

- **Widgets** : GTK_Fixed
- **Package** : GTK_Fixed
- **Descendance** : GTK_Widget >> GTK_Container
- **Description** : Conteneur de taille infinie permettant de positionner divers widgets à partir de coordonnées.

Utilisation des GTK_Fixed

Ce sera certainement le widget le plus simple de ce chapitre. Son constructeur se nomme simplement `Gtk_New()` et n'a aucun paramètre particulier. Pour positionner un widget à l'intérieur d'un `GTK_Fixed`, la méthode à employer s'appelle simplement `put()`. Ses paramètres sont simplement les coordonnées ($x; y$) du coin supérieur gauche de votre widget.

Exemple :

Code : Ada

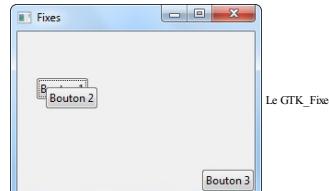
```
WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Enums ; USE Gtk.Enums ;
WITH Gtk.Button ; USE Gtk.Button ;
WITH Gtk.Fixed ; USE Gtk.Fixed ;

PROCEDURE MaFenetre IS
    Win : Gtk.Window ;
    Couche : Gtk.Fixed ;
    Btn1, Btn2, Btn3 : Gtk.Button ;
BEGIN
    Init ;
    Gtk_New(Win, Window_Toplevel) ;
    Win.Set_Title("Fixes") ;
    Win.set_default_size(150, 100) ;

    Gtk_New(Couche) ;
    Win.Add(Couche) ;

    Gtk_New(Btn1, "Bouton 1") ; Couche.put(Btn1, 20, 50) ;
    Gtk_New(Btn2, "Bouton 2") ; Couche.put(Btn2, 30, 60) ;
    Gtk_New(Btn3, "Bouton 3") ; Couche.put(Btn3, 200, 150) ;

    Win.Show_all ;
    Main ;
END MaFenetre ;
```



Le GTK_Fixed

Vous remarquerez deux choses : ce conteneur autorise le chevauchement de widgets. C'est à vous de faire en sorte qu'il n'y ait pas de souci d'affichage, ce qui est plus contraignant qu'avec les boîtes ou les tables. Ensuite, le Bouton n°3 aurait dû être en dehors de la fenêtre ; en effet, celle-ci a une taille de 150×100 alors que le bouton 3 est situé au point $(200; 150)$, soit en dehors. Mais le `GTK_Fixed` a agrandi automatiquement la fenêtre pour pouvoir l'afficher. Il existe un conteneur similaire, les `GTK_Layout`, qui fonctionne de la même façon mais pour lequel la fenêtre ne serait pas agrandie si elle ne s'embêtera pas avec cela.

Enfin, si l'un de vos widgets devait être déplacé, il suffirait d'utiliser la méthode `move()` qui fonctionne comme `put()`.

En résumé :

- Utilisez les `GTK_Alignment` pour positionner finement un unique widget.
- Pour positionner plusieurs widgets, utilisez les `GTK_Box`, `GTK_Button_Box` ou `GTK_Table`.
- Si vous avez besoin de positionner manuellement vos widgets (au pixel près) optez pour les `GTK_Fixed` mais pensez que le rendu ne sera pas forcément le même sur un autre ordinateur.
- N'hésitez pas à imbriquer divers conteneurs les uns dans les autres pour obtenir la présentation souhaitée.

Les signaux

Maintenant que vous savez créer des fenêtres, les organiser, y ajouter des boutons ou des étiquettes, il est temps que tout cela agisse un petit peu. À quoi bon cliquer sur des boutons si rien ne se passe ? Nous allons devoir tout d'abord nous intéresser à la gestion bas niveau des événements pour comprendre le fonctionnement de GTK. Puis nous retournerons au haut niveau et à la programmation événementielle à proprement parler.

Ce chapitre est fondamental pour la suite, mais il est ardu. N'hésitez pas à le relire ou à revenir plus tard car il faut être précis sur les types ou sous-packages employés (et ils sont nombreux). GNAT sera impitoyable si vous ne respectez pas scrupuleusement mes conseils.

Le principe

Les problèmes de la programmation événementielle



On serait tenté d'écrire ce genre de chose :

Code : Ada

```
IF Bouton1.Is_clicked
  THEN Ma_Procedure ;
END IF ;
```

Le souci, c'est que tout notre code consiste à créer et paramétrier des widgets, les afficher et enfin lancer une boucle infinie (`Main`). Où écrire ces quelques lignes ? Le programme ne met que quelques millisecondes à créer les widgets et à les afficher, cela ne laisse pas vraiment le temps à l'utilisateur d'aller cliquer sur un bouton. Et puis un clic ne dure pas plus d'une seconde, comment faire pour que le programme exécute cette ligne de test durant cette fraction de seconde ? C'est impossible.

Le problème est même plus profond que cela. Nous codions jusqu'ici de manière linéaire. Certes, nous avons les conditions pour exécuter telle ou telle instruction, les boucles pour répéter ou pas certaines portions de code, les tâches pour exécuter plusieurs programmes en même temps. Mais tout cela restait très prévisible : l'utilisateur ne pouvait sortir du cadre très fermé que vous lui aviez concocté à grands coups de `IF`, `CASE`, `LOOP`, `TASK`... Avec nos programmes fenêtrés, nous ne pouvons savoir quand l'utilisateur cliquera sur notre bouton, ni si le fera, ni même s'il aura cliqué sur d'autres widgets auparavant. Vous êtes en réalité dans l'inconnu.



Une boucle infinie ? C'est une idée. Mais nous disposons déjà de ce genre de boucle : `Main` ! `Main` contrôle en permanence les différents widgets créés et vérifie s'ils ne sont pas cliqués, déplacés, survolés... Bref, `Main` attend le moindre événement pour réagir.

Le principe Signal-Procédure de rappel

Pour comprendre comment faire agir nos widgets, vous devez comprendre le principe de fonctionnement bas-niveau de GTK. Lorsqu'un événement se produit, par exemple lorsque l'utilisateur clique sur un bouton, un signal est émis.



Aussitôt, ce signal est intercepté par `Main` qui va associer le widget et le type de signal émis à une procédure particulière, appelée **procédure de rappel** ou **callback**.



Pour que votre programme puisse faire le lien, il faut que vous ayez précédemment connecté le widget en question et son signal à cette fameuse procédure de rappel. C'est ce à quoi nous allons nous attacher maintenant.

Connecter un signal à un callback

Fermant proprement le programme

Code de base et package

Notre premier projet est simple : créer une fenêtre avec un unique bouton. Lorsque l'utilisateur cliquera sur le bouton ou sur le bouton de fermeture de la fenêtre, cela mettra fin au programme. Pour cela nous n'aurons pas à créer de procédure de rappel, nous ferons appel à des procédures préexistantes. Voici donc notre code de base (que vous devriez pouvoir déchiffrer seul désormais) :

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ;          USE Gtk.Enums ;
WITH Gtk.Button ;         USE Gtk.Button ;

PROCEDURE MaFenetre IS
  Win    : Gtk.Window ;
  Btn1   : Gtk_Button ;
BEGIN
  Init ;                      --Initialisation de GTK
  Gtk_New(Win,Window_Toplevel) ; --Création et paramétrage
  de la fenêtre
  Win.Set_Title("Fenêtre") ;
  win.set_default_size(200,150) ;

  Gtk_New_With_Mnemonic(Btn1, "_Exit") ; --Création d'un bouton et
  ajout à la fenêtre
  win.add(Btn1) ;

  Win.Show_all ;               --Affichage et lancement
  de la boucle d'événements
  Main ;
END MaFenetre ;
```

Pour manipuler nos divers widget, nous aurons besoin du package `Gtk.Handlers` (pour rappel, le verbe anglais « to handle » signifie « manipuler »). Attention ce package en lui-même ne contient presque rien d'utilisable. Si vous ouvrez ses spécifications, vous vous rendrez compte qu'il contient en fait tout un ensemble d'autres sous-packages génériques. Je vous expliquerai les détails plus tard. Sachez pour l'instant que nous avons besoin du sous-package générique `Gtk.Handlers.Callback`. Il n'a qu'un seul paramètre de généralité : le type `Widget_Type` dont le nom me semble assez clair.

Nous devons donc l'instancier en créant un nouveau package `P_Callback`. Pour l'instant, nous n'allons fournir en paramètre que le type `GTK_Window_Record` et nous verrons pour le bouton un peu plus tard.

Code : Ada

```
PACKAGE P_Callback IS NEW Gtk.Handlers.Callback(Gtk.Window_Record) ;
```

```
USE P_Callback;
```



Attention, le type fourni est bien `Gtk.Window_Record` et pas `Gtk.Window` qui est, rappelons-le, un type pointeur !

La procédure de connexion

Vous devriez trouver à la ligne 847 de `Gtk.Handlers` les spécifications d'une procédure essentielle :

Code : Ada

```
procedure Connect
  ( Widget : access Widget_Type'Class;
    Name : Glib.Signal_Name;
    Cb : Simple_Handler;
    After : Boolean := False);
```

C'est cette procédure qui va connecter le widget et le signal qu'il émet avec une procédure de votre choix. Il est important de bien regarder sa spécification si vous ne voulez pas que GNAT soit désagréable avec vous. Tout d'abord le widget : on attend un pointeur sur une classe de widget. Autrement dit, dans notre cas présent, un objet de type `Gtk.Window` (et pas `Gtk.Window_Record`). Ensuite, le paramètre `Name` correspond au nom du signal émis. Ce nom est en fait un simple `string` que je vous donnerai. Le dernier paramètre (`After`) ne nous servira que si vous voulez imposer que la connexion se fasse qu'une fois toutes les connexions par défaut effectuées. Bref, aucun intérêt à notre niveau. Enfin, nous arrivons au paramètre `Cb` de type `Simple_Handler`. Ce type correspond à un pointeur sur une procédure prenant en paramètre un pointeur sur une classe de widget (le widget grâce auquel vous avez instancié `Gtk.Handlers.Callback`). C'est donc ce paramètre qui enregistrera la procédure à connecter.



Vous devrez être le plus rigoureux possible lorsque vous utiliserez la procédure de connexion. Le moindre écart posera souci à la compilation ! Alors n'hésitez pas à revenir lire cette partie plusieurs fois.

La procédure de rappel

Nous allons créer une procédure pour sortir de la boucle d'événements : `Stop_Program`. Cette procédure utilisera l'instruction `Main_Quit` qui met fin à `Main`. La logique voudrait que cette procédure n'ait aucun paramètre. Pourtant, si j'ai préalablement attiré votre attention sur la procédure `connect()`, c'est pour que vous compreniez comment définir votre callback. `connect()` va non seulement connecter un signal à une procédure, mais il transmettra également le widget appelant à ladite procédure si ce signal est détecté. Nous voilà donc obligés d'ajouter un paramètre à notre callback. Quant à son type, il suffit de regarder les spécifications de `connect()` pour le connaître : « `ACCESS Widget_Type'Class` » ! Ce qui nous donnera :

Code : Ada

```
PROCEDURE Stop_Program(Emetteur : access Gtk.Window_Record'class) IS
BEGIN
  Main_Quit;
END Stop_Program;
```

Si vous compiler, GNAT ne cessera de vous dire : "warning : formal parameter "Emetteur" is not referenced". Ce n'est pas une erreur rédhibitoire mais c'est toujours gênant de laisser traîner ces avertissements. Alors pour indiquer à GNAT que le paramètre `Emetteur` est sciemment inutilisé, vous pourrez ajouter le `PRAGMA unreferenced` :

Code : Ada

```
PROCEDURE Stop_Program(Emetteur : access Gtk.Window_Record'class) IS
PRAGMA Unreferenced (Emetteur);
BEGIN
  Main_Quit;
END Stop_Program;
```

La connexion, enfin !

Le package est instancié et le callback rédigé. Il ne reste plus qu'à effectuer la connexion à l'aide de la procédure vue plus haut. Pour cela, vous devez connaître le nom du signal de fermeture de la fenêtre : "`destroy`". Ce signal est envoyé par n'importe quel widget lorsqu'il est détruit. Notre code donne donc :

Code : Ada

```
WITH Gtk.Main;
WITH Gtk.Window;
WITH Gtk.Enums;
WITH Gtk.Button;
WITH Gtk.Handlers;
WITH P_Callback;

PROCEDURE MaFenetre IS
  --Instantiation du package pour la connexion
  PACKAGE P_Callback IS NEW Gtk.Handlers.Callback(Gtk.Window_Record);
  USE P_Callback;
  --Notre callback
  PROCEDURE Stop_Program(Emetteur : access Gtk.Window_Record'class) IS
  PRAGMA Unreferenced (Emetteur);
  BEGIN
    Main_Quit;
  END Stop_Program;

  Win    : Gtk.Window;
  Btn1  : Gtk_Button;
BEGIN
  Init;
  Gtk_New(Win, Window_Toplevel);
  Win.Set_Title("Fenêtre");
  Win.set_default_size(200,150);

  Gtk_New_With_Mnemonic(Btn1, "_Exit");
  win.add(Btn1);

  --Connexion du signal "destroy" avec le callback Stop_program
  --Pensez que le signal est un string, non un type énuméré
  --N'oubliez pas que le 3e paramètre doit être un pointeur
  connect(win, "destroy", Stop_Program'access);
  --en cas de souci, essayez ceci :
  --connect(win, "destroy", to_Marshaller(Stop_Program'access));
;
  Win.Show_all;
  Main;
END MaFenetre;
```

Utiliser le bouton

Nous voudrions maintenant pouvoir faire de même avec notre bouton. Le souci, c'est que la procédure `connect()` a pour nom complet `P_Callback.connect()` et que le package `P_Callback` a été instancié pour les objets de la classe `Gtk.Window_Record'Class`. Deux solutions s'offrent à nous : soit créer un second package `P_Callback_Button`, soit améliorer le premier.

Opte pour la deuxième méthode. Il suffit d'utiliser une classe commune aux `Gtk.Window_Record` et aux `Gtk.Button_Record`. Vous savez désormais que fenêtres et boutons sont deux widgets. Nous pourrions donc utiliser la classe `GTK_Widget_Record'Class` et citer le package `Gtk.Widget`. Mais nous savons également que ce sont deux conteneurs (`GTK.Container_Record'Class`) et même des conteneurs pour un seul élément (`GTK_Bin_Record'Class`). Nous pouvons donc généraliser notre package de trois façons différentes. Les voici, de la plus restrictive à la plus large :

Code : Ada

```
WITH Gtk.Bin;
...
PACKAGE P_Callback IS NEW Gtk.Handlers.Callback(Gtk_Bin_Record);
USE P_Callback;

PROCEDURE Stop_Program(Emetteur : access Gtk_Bin_Record'class) IS
```

```

...
-- OU

WITH Gtk.Container ;           USE Gtk.Container ;
...
PACKAGE P_Callback IS          NEW
Gtk.Handlers.Callback(Gtk.Container_Record) ;
USE P_Callback ;

PROCEDURE Stop_Program(Emetteur : access Gtk.Container_Record'class)
IS ...

-- OU

WITH Gtk.Widget ;             USE Gtk.Widget ;
...
PACKAGE P_Callback IS NEW Gtk.Handlers.Callback(Gtk_Widget_Record) ;
USE P_Callback ;

PROCEDURE Stop_Program(Emetteur : access Gtk_Widget_Record'class) IS
...

```

Ne reste plus qu'à effectuer la connexion. Si peu de signaux sont intéressants pour l'instant avec les fenêtres, avec les boutons, nous avons davantage de choix. Voici quelques signaux :

- "clicked": le bouton a été cliqué, c'est-à-dire enfoncé puis relâché.
- "pressed": le bouton a été enfoncé
- "released": le bouton a été relâché
- "enter": la souris survole le bouton
- "leave": la souris ne survole plus le bouton

Voici ce que donnerait notre programme :

Code : Ada

```

WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ;          USE Gtk.Enums ;
WITH Gtk.Button ;         USE Gtk.Button ;
WITH Gtk.Bin ; USE Gtk.Bin ;
WITH Gtk.Handlers ;

PROCEDURE MaFenetre IS
PACKAGE P_Callback IS NEW Gtk.Handlers.Callback(Gtk_Bin_Record) ;
USE P_Callback ;

PROCEDURE Stop_Program(Emetteur : access Gtk_Bin_Record'class) IS
PRAGMA Unreferenced (Emetteur);
BEGIN
  Main_Quit;
END Stop_Program;

BEGIN
  Init ;
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Fenêtre") ;
  Win.set_default_size(200,150) ;

  Gtk_New(With Mnemonic(Btn1, "_Exit") ;
  Win.Add(Btn1) ;

  Connect(Win, "destroy", To_Marshaller(Stop_Program'ACCESS)) ;
  Connect(Btn1, "clicked", To_Marshaller(Stop_Program'ACCESS)) ;

  Win.Show_all ;
  Main ;
END MaFenetre ;

```

Interagir avec les widgets

Un callback à deux paramètres



Fermer le programme, c'est facile : il suffit d'une seule instruction. Mais comment je fais si je veux créer des callbacks modifiant d'autres widgets ?

Pour y voir clair, nous allons créer un programme «Hello world». L'idée est simple : notre fenêtre affichera une étiquette nous saluant uniquement si nous appuyons sur un bouton. Nous avons donc besoin de créer une fenêtre contenant une étiquette, un bouton et bien sûr un conteneur :

Code : Ada

```

WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Enums ;          USE Gtk.Enums ;
WITH Gtk.Button ;         USE Gtk.Button ;
WITH Gtk.Label ;          USE Gtk.Label ;
WITH Gtk.Box ;             USE Gtk.Box ;
WITH Gtk.Widget ;          USE Gtk.Widget ;
WITH Gtk.Handlers ;

PROCEDURE MaFenetre IS
--Sous packages pour manipuler les callbacks
PACKAGE P_Callback IS          NEW
Gtk.Handlers.Callback(Gtk_Widget_Record) ;
USE P_Callback ;

--CALLBACKS
PROCEDURE Stop_Program(Emetteur : access Gtk_Widget_Record'class)
IS
  PRAGMA Unreferenced (Emetteur);
BEGIN
  Main_Quit;
END Stop_Program;

--WIDGETS
Win    : Gtk.Window ;
Btn    : Gtk.Button ;
Lbl    : Gtk.Label ;
Box   : Gtk_Vbox ;
BEGIN
  Init ;

  --CREATION DE LA FENETRE
  Gtk_New(Win,Window_Toplevel) ;
  Win.Set_Title("Fenêtre") ;
  Win.set_default_size(200,150) ;

  --CREATION DU BOUTON ET DE L'ETIQUETTE
  Gtk_New(With Mnemonic(Btn, "Dis _bonjour !") ;
  Gtk_New(Lbl,"")) ;

  --CREATION DE LA BOITE
  Gtk_New_Vbox(Box) ;
  Box.Pack_Start(Lbl) ;
  Box.Pack_Start(Btn) ;
  Win.Add(Box) ;

  --CONNEXION AVEC LE CALLBACK DE FERMETURE
  Connect(Win, "destroy", Stop_Program'ACCESS) ;

  Win.Show_all ;
  Main ;
END MaFenetre ;

```

Ce code n'est pas très compliqué, vous devriez pouvoir le comprendre seul. Maintenant, nous allons créer un callback supplémentaire appelé `Dis_Bonjour()` qui modifiera une étiquette (ne nous préoccupons pas de la connexion pour l'instant) :

Code : Ada

```
PROCEDURE Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class ;
                      Label : Gtk_Label) IS
  PRAGMA Unreferenced(Emetteur) ;
BEGIN
  IF Label.Get_Text'Length = 0
  THEN Label.Set_Text("Hello world !");
  ELSE Label.Set_Text("Ouais, ouais... salut !");
  END IF ;
END Dis_Bonjour;
```

J'utilise deux méthodes de la classe `GTK_Label_Record:Set_Text()` qui modifie le texte de l'étiquette et `Get_Text()` qui renvoie le contenu. Rien de bien compliqué mais un souci toutefois : nous avons besoin de deux paramètres !

Un nouveau package de callbacks

Or, pour l'instant notre package `P_Callback` n'est instancié qu'avec un seul type : la classe `GTK_Widget_Record` du widget émetteur. Cela signifie que nos callbacks ne peuvent avoir qu'un unique paramètre de ce même type. Comment faire pour accéder à un second widget ? Plusieurs solutions s'offrent à vous. L'une d'elles consiste à placer nos widgets dans un package spécifique pour en faire des variables globales et ainsi ne plus avoir à les déclarer comme paramètre formel de nos méthodes. Vous pourrez ainsi facilement accéder à votre `GTK_Label`. Toutefois, vous savez que je n'aime pas abuser des variables globales. D'autant plus que les `GTK_Widget`, `GTK_Button` ou `GTK_Label` sont, vous le savez des pointeurs généralisés : nous multiplions les prises de risque.

Pour résoudre ce dilemme, `GtkAda` fournit toute une série de package pour nos callbacks. L'un d'entre eux, `Gtk.Handlers.User_Callback` permet de passer un paramètre supplémentaire à nos procédures de connexion (il faut donc l'instancier à l'aide de deux types). Ces deux paramètres sont simplement ceux fournis à notre callback : `Gtk_Widget_Record` pour l'émetteur (le premier paramètre de notre callback est un pointeur vers ce type) ; `Gtk_Label` pour le second paramètre. D'où la définition :

Code : Ada

```
PACKAGE P_UCallback IS
  NEW
  Gtk.Handlers.User_Callback(Gtk_Widget_Record,Gtk_Label) ;
  USE P_Ucallback ;
```

La connexion

La connexion se fait de la même manière que précédemment : en utilisant la procédure `connect()`. Toutefois celle-ci admettra un paramètre supplémentaire :

Code : Ada

```
procedure Connect(
  Widget : access Widget_Type'Class;
  Name   : Glib.Signal_Name;
  Marsh  : Marshallsers.Marschaller;
  User_Data : User_Type;
  After   : Boolean := False);
```

On retrouve les paramètres habituels, sauf `User_Data` qui ici correspond au type `GTK_Label` employé par votre callback. Voici ce que donne notre code :

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;        USE Gtk.Window ;
WITH Gtk.Enums ;         USE Gtk.Enums ;
WITH Gtk.Button ;        USE Gtk.Button ;
WITH Gtk.Label ;          USE Gtk.Label ;
WITH Gtk.Box ;            USE Gtk.Box ;
WITH Gtk.Widget ;         USE Gtk.Widget ;
WITH Gtk.Handlers ;
```

--SOUS PACKAGES POUR MANIPULER LES CALLBACKS

```
PACKAGE P_Callback IS
  NEW
  Gtk.Handlers.Callback(Gtk_Widget_Record) ;
  USE P_Callback ;
```

--CALLBACKS

```
PROCEDURE MaFenetre IS
  PROCEDURE Stop_Program(Emetteur : access Gtk_Widget_Record'class)
  IS
    PRAGMA Unreferenced(Emetteur) ;
  BEGIN
    Main.Quit;
    END Stop_Program;
```

```
PROCEDURE Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class ;
                      Label : GTK_Label) IS
  PRAGMA Unreferenced(Emetteur) ;
BEGIN
  IF String(Label.Get_Text)'Length = 0
  THEN Label.Set_Text("Hello world !");
  ELSE Label.Set_Text("Ouais, ouais... salut !");
  END IF ;
END Dis_Bonjour;
```

--WIDGETS

```
Win   : Gtk.Window ;
Btn   : Gtk_Button ;
Lbl   : Gtk_Label ;
Box   : Gtk_Vbox ;
BEGIN
  Init ;
```

--CREATION DE LA FENETRE

```
Gtk_New(Win,Window_Toplevel) ;
Win_Set_Title("Fenetre") ;
win.set_default_size(200,150) ;
```

--CREATION DU BOUTON ET DE L'ETIQUETTE

```
Gtk_New(Btn,Window_Toplevel) ;
Gtk_New_With_Mnemonic(Btn, "Dis_bonjour !");
Gtk_New(Lbl,"") ;
```

--CREATION DE LA BOITE

```
Gtk_New_Vbox(Box) ;
Box_Pack_Start(Lbl) ;
Box_Pack_Start(Btn) ;
Win.Add(Box) ;
```

--CONNEXION AVEC LES CALLBACKS

```
Connect(Win, "destroy", Stop_Program'ACCESS) ;
Connect(Btn, "clicked", Dis_Bonjour'ACCESS, Lbl) ;
```

```
Win.Show_all ;
Main ;
```

```
END MaFenetre ;
```



Perfectionner encore notre code

Mais... ça veut dire que si je voulais ajouter un deuxième bouton qui modifierait un autre widget (une image par exemple), il faudrait créer un nouveau sous package du genre : `PACKAGE P_UCallback2 IS NEW Gtk.Handlers.User_Callback(Gtk_Widget_Record, On_Autre_Type_De_Widget) ;` Ça va devenir encombrant à la longue tous ces packages. (2)

Vous avez raison, nous devons optimiser notre code pour ne pas créer plusieurs instances d'un même package. Et pour cela, nous allons réutiliser les types contrôlés ! Si, comme je vous l'avais conseillé, vous placez tous vos widgets dans un type contrôlé, accompagné d'une méthode `Initialize()`, vous n'aurez dès lors plus qu'une seule instantiation à effectuer :

Code : Ada - P_Fenetre.ads

```
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Enums ; USE Gtk.Enums ;
WITH Gtk.Button ; USE Gtk.Button ;
WITH Gtk.Label ; USE Gtk.Label ;
WITH Gtk.Box ; USE Gtk.Box ;
WITH Gtk.Widget ; USE Gtk.Widget ;
WITH Ada.Finalization ; USE Ada.Finalization ;
WITH Gtk.Handlers ;

PACKAGE P_Fenetre IS
  TYPE T_Fenetre IS NEW Controlled WITH RECORD
    Win : Gtk.Window ;
    Btn : Gtk.Button ;
    Lbl : Gtk.Label ;
    Box : Gtk.Vbox ;
  END RECORD ;

  PROCEDURE Initialize(F : IN OUT T_Fenetre) ;

  PACKAGE P_Handlers IS
    NEW Gtk.Handlers.Callback(Gtk.Widget_Record) ;
    USE P_Handlers ;
  PACKAGE P_UHandlers IS
    NEW Gtk.Handlers.User_Callback(Gtk.Widget_Record,T_Fenetre) ;
    use P_UHandlers ;
  END P_Fenetre ;
```

Code : Ada - P_Fenetre.adb

```
WITH Gtk.Main ; USE Gtk.Main ;
WITH P_Callbacks ; USE P_Callbacks ;

PACKAGE BODY P_Fenetre IS
  PROCEDURE Initialize(F : IN OUT T_Fenetre) IS
  BEGIN
    Init ;
    Gtk_New(GTK.Window(F.Win),Window_Toplevel) ;
    F.Win.Set_Title("Fenetre") ;
    F.Win.set_default_size(200,150) ;

    Gtk_New_With_Mnemonic(F.Btn, "Dis_bonjour !");
    Gtk_New(F.Lbl,"") ;

    Gtk_New_Vbox(F.Box) ;
    F.Box.Pack_Start(F.Lbl) ;
    F.Box.Pack_Start(F.Btn) ;
    F.Win.Add(F.Box) ;

    Connect(F.Win, "destroy", Stop_Program'ACCESS) ;
    Connect(F.Btn, "clicked", Dis_Bonjour'ACCESS, F) ;

    F.Win.Show_all ;
  END Initialize ;
END P_Fenetre ;
```

Code : Ada - P_Callbacks.ads

```
WITH Gtk.Widget ; USE Gtk.Widget ;
with P_Fenetre ; use P_Fenetre ;

PACKAGE P_Callbacks IS
  PROCEDURE Stop_Program(Emetteur : access Gtk_Widget_Record'class)
  ;
  PROCEDURE Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class ;
  F : T_Fenetre) ;
END P_Callbacks ;
```

Code : Ada - P_Callbacks.adb

```
WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Label ; USE Gtk.Label ;

PACKAGE BODY P_Callbacks IS
  PROCEDURE Stop_Program(Emetteur : access Gtk_Widget_Record'class)
  IS
    PRAGMA Unreferenced (Emetteur) ;
  BEGIN
    Main_Quit;
  END Stop_Program ;

  PROCEDURE Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class ;
  F : T_Fenetre) IS
    PRAGMA Unreferenced(Emetteur) ;
  BEGIN
    IF F.Lbl.Get_Text'Length = 0
      THEN F.Lbl.Set_Text("Hello world !");
      else F.Lbl.Set_Text("Ouais, ouais ... salut !");
    END IF ;
  END Dis_Bonjour;
END P_Callbacks ;
```

Code : Ada - MaFenetre.adb

```
WITH Gtk.Main ; USE Gtk.Main ;
WITH P_Fenetre ; USE P_Fenetre ;

PROCEDURE MaFenetre IS
```

```
F : T_Fenetre ;
pragma Unreferenced(F) ;
BEGIN
  Main ;
END MaFenetre ;
```

Autres packages de callback



Le package `Gtk.Handlers` comprend 6 sous-packages de callback :

- `Return_Callback` : ce package permet d'employer des fonctions comme callbacks et pas des procédures. Elle prend en paramètre le type de widget émetteur, bien entendu, ainsi que le type des résultats retournés par les fonctions.
- `User_Return_Callback` : similaire au package `User_Callback` utilisé dans cette sous-partie, mais pour des fonctions de rappel, il prend en paramètre la classe du widget émetteur, le type des résultats retournés par les fonctions de rappel, ainsi que le type `User_Type` du paramètre supplémentaire (notez `GTK_Label` dans l'exemple ci-dessus).
- `User_Return_Callback_With_Setup` : similaire au package précédent, il permet toutefois de fournir une procédure d'initialisation des objets de type `User_Type`.
- `Callback` : vu dans la sous-partie n°2.
- `User_Callback` : vu dans la sous-partie n°3.
- `User_Callback_With_Setup` : similaire au package `User_Callback` mais fournit une procédure d'initialisation des objets de type `User_Type`.

GDK et les événements

Le clic droit

Les signaux de GTK.Widget



Les signaux dont je vous ai parlé jusque là sont ceux liés aux boutons, on les trouve dans le package `Gtk.Button`. Mais aucun signal lié au `GTK_Button` ne prend en compte le clic-droit. Il faut donc chercher un peu plus dans les tréfonds de GTK pour y parvenir. Tout d'abord, nous savons que les boutons ne sont rien d'autre que des widgets, ou pour parler le langage de GTK, ce sont des `GTK_Widget`. Grâce aux principes de la POO, nous savons que les `GTK_Button` héritent par dérivation des méthodes des `GTK_Widget`. Nous avons donc qu'à fouiner dans le package `GTK_Widget` pour y trouver ce que nous cherchons. Malheureusement, les seuls signaux que nous y trouvons et qui soient liés à la souris sont les suivants :

Code : Ada

```
Signal_Button_Press_Event      : constant Glib.Signal_Name := "button_press_event";
Signal_Button_Release_Event    : constant Glib.Signal_Name := "button_release_event";
```

Le premier est émis lorsqu'un bouton de la souris est enfoncé, le second lorsqu'il est relâché. Mais ça ne nous avance pas sur la question : est-ce un clic gauche ou droit ? Sans compter que certaines souris ont plus de 2 boutons ! C'est pourtant un début. Nous allons donc modifier la connexion entre notre bouton et la procédure `Dis_Bonjour` en utilisant ce signal :

Code : Ada - P_Fenetre.adb

```
Connect(Btn, "button_press_event",     Dis_Bonjour'ACCESS, F);
OR
Connect(Btn, Signal_Button_Press_Event, Dis_Bonjour'ACCESS, F);
```

User_Return_Callback, nous voilà !

GNAT compile sans souciller mais lorsque nous lançons notre programme, une jolie exception est levée :



Autrement dit, pour les non anglophones, avec ce signal et ce type de widget nous ne pouvons utiliser comme callback que des fonctions ! Et pour être plus précis, des fonctions renvoyant un booléen. En effet, GTK souhaite savoir s'il doit jouer ou non la petite animation du bouton qui s'enfonce lors du clic sur le `GTK_Button`. Cela remet en cause une bonne partie de notre code et notamment le package utilisé. Nous allons devoir abandonner le package `Gtk.Handlers.User_Callback` pour le package `Gtk.Handlers.User_Return_Callback`.

Code : Ada - P_Fenetre.ads

```
PACKAGE P_UHandlers IS
  Gtk.Handlers.User_Return_Callback(Gtk_Widget_Record, Boolean,
  T_Fenetre) ;
  USE P_UHandlers ;
```

Le premier paramètre pour instancier notre package est toujours le type du widget émetteur, le second correspond au type du résultat retourné par les callbacks, enfin le dernier type correspond au type du paramètre de la fonction de callback. Il nous faut désormais en venir à notre procédure et la transformer en fonction adéquate :

Code : Ada - P_Callback.ads

```
FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class ;
                      F      : T_Fenetre) RETURN Boolean ;
```

Code : Ada - P_Callback.adb

```
FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class ;
                      F      : T_Fenetre) RETURN Boolean IS
  PRAGMA Unreferenced(Emetteur) ;
BEGIN
  IF F.Lbl.Get_Text'Length = 0
    THEN F.Lbl.Set_Text("Hello world !");
    ELSE F.Lbl.Set_Text("Ouais... salut !");
  END IF ;
  RETURN True ;
END Dis_Bonjour;
```

Compilez et lancez votre programme : tout fonctionne de nouveau, à un détail près. Regardez attentivement le bouton «Dis bonjour» et cliquez dessus. Certes le texte s'affiche convenablement au-dessus, mais le bouton reste immobile ! Comme si ce n'était qu'une simple image. La raison vient de la ligne 20 du fichier `P_Callback.adb` : nous avons renvoyé `TRUE`. GTK considère que tout s'est bien passé durant notre callback et que celui-ci s'occupe de tout, notamment de l'aspect du bouton. Remplacez cette ligne par `RETURN FALSE` ; et vous verrez que ce détail s'arrangera.

Distinguer gauche et droite



Pas tout à fait. Si vous testez votre programme, vous vous apercevez que désormais, un clic droit activera le callback `Dis_Bonjour` tout aussi bien qu'un clic gauche ou qu'un clic avec un autre bouton de la souris (la molette par exemple). Un clic droit active donc notre callback, il ne nous reste plus qu'à distinguer la gauche et la droite.

Pour ce faire, nous allons continuer notre exploration de GTK et même parcourir les bibliothèques de GDK ! Plus exactement, nous allons avoir besoin du package `GDK_Event.ads` que je vous conseille d'ouvrir pour suivre la fin de ce chapitre. À la ligne 229 de ce même package, vous trouverez le type suivant :

Code : Ada

```
type Gdk_Event is new Gdk.C_Proxy;
```

Les `GDK_Event` sont des événements. Pour simplifier, disons qu'il s'agit de la version bas niveau des signaux. Lorsqu'un `GDK_Event` est émis, la boucle Main l'interprète et émet le signal correspondant. L'idée serait donc d'analyser l'événement qui a émis le `Signal_Button_Press_Event` à défaut de l'intercepter nous-même. Nous allons donc ajouter un nouveau paramètre à notre callback : `Evenement` de type `GDK_Event`.

Code : Ada - P_Callbacks.ads

```
WITH Gtk_Widget; USE Gtk_Widget;
WITH P_Fenetre; USE P_Fenetre;
WITH GDK_Event; USE GDK_Event;

PACKAGE P_Callbacks IS
    PROCEDURE Stop_Program(Emetteur : access Gtk_Widget_Record'Class);
    FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class;
        Evenement : GDK_Event; F : T_Fenetre) RETURN Boolean;
END P_Callbacks;
```

Vous vous demandez comment nous allons pouvoir récupérer cet événement ? Soyez patients, faites comme si ce problème n'en était pas un et peu dans nos packages. Nous souhaiterions connaître quel bouton a été cliqué ? Une fonction `Get_Button()` existe à la ligne 418 de `Gdk-event.ads`. Celle-ci analyse un événement et renvoie le numéro du bouton qui a été cliqué. Venons-en au corps de notre callback :

Code : Ada - p_callbacks.adb

```
FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class;
    Evenement : GDK_Event; F : T_Fenetre) RETURN Boolean IS
    PRAGMA Unreferenced(Emetteur);
BEGIN
    CASE Get_Button(Evenement) IS
    WHEN 1 => F.Lbl.Set_Text("Hello world !");
    WHEN 3 => F.Lbl.Set_Text("Ouais, ouais... salut !");
    WHEN OTHERS => F.Lbl.Set_Text("");
    END CASE;
    RETURN False;
END Dis_Bonjour;
```

Désormais, notre programme affichera "Hello world !" lors d'un clic gauche et "Ouais, ouais... salut !" lors d'un clic droit. Un clic avec tout autre bouton, comme la molette, effacera le contenu de l'étiquette.

Connaitre l'événement grâce aux Marshallers

 Mais ??? GNAT n'accepte plus la connexion entre mon `GTK_Button` et mon callback ? Comment lui dire qu'il y a un paramètre en plus pour `Dis_Bonjour` ? Et quand est-ce que l'on peut obtenir ce `GDK_Event` pour de vrai ?

Pas de panique. On ne va pas chercher à obtenir cet événement puisqu'il a déjà été capturé afin d'émettre le signal `Signal_Button_Press_Event`. lorsque notre callback sera déclenché, il sera trop tard pour obtenir l'événement et si nous cherchons à le capturer, cela empêchera l'exécution du callback. La seule solution est de faire appel aux fonctions `To_Marshaller()`. Elles vont nous permettre de transformer un simple pointeur sur fonction en un Marshaller, type qui permet d'embarquer davantage d'informations. Vous n'y comprenez rien, alors voyez l'exemple suivant :

Code : Ada - P_Fenetre.adb

```
Connect(F.Btn, "button_press_event",
        To_Marshaller(Dis_Bonjour'ACCESS), F);
```

Vous remarquerez que je n'ai ajouté aucun paramètre pour le `GDK_Event`. J'ai simplement converti mon pointeur sur `Dis_Bonjour` en Marshaller. Testez ce code et dites-moi des nouvelles.

Le double clic

Le `GDK_Event_Type`

 Et pour un double clic, on fait comment ?

Bonne question. Jetez un oeil aux toutes premières lignes du package `GDK-Event.ads`. Vous allez y trouver un type très intéressant, `Gdk_Event_Type`, dont voici la liste des valeurs :

Code : Ada

```
type Gdk_Event_Type is (Nothing, Delete, Destroy, Expose, Motion_
Notify, Button_Press, Gdk_2button_Press, Gdk_3button_Press,
Button_Release, Key_Press, Key_Release, Enter_Notify, Leave_Notify,
Focus_Change, Configure,
Map, Unmap, Property_Notify,
Selection_Clear, Selection_Request,
Selection_Notify, Proximity_In,
Proximity_Out, Drag_Enter,
Drag_Leave, Drag_Motion, Drag_Status,
Drop_Start, Drop_Finished,
Client_Event, Visibility_Notify, No_Expose,
Scroll, Window_State,
Setting, Owner_Change, Grab_Broken);
```

Il s'agit des différents types d'événements de bases qu'un `GDK_Event` peut prendre. Nous n'allons pas tous les voir mais si vous êtes attentif vous y trouverez `button_Press`, l'événement pour un simple clic de souris, `Gdk_2button_Press`, l'événement pour un double clic de souris, `Gdk_3button_Press`, pour un triple clic, `Button_Release`, pour un bouton de souris relâché, `Key_Press`, pour une touche de clavier enfoncée ou `Key_Release`, pour une touche de clavier relâchée. Nous verrons les deux dernières juste après : concentrons-nous sur `Gdk_2button_Press` et `Gdk_3button_Press`.

Nous souhaiterions que notre callback affiche une phrase en espagnol lors d'un double clic. Nous devons donc distinguer les clics simples des clics doubles (le principe sera le même pour les triples clics) et pour cela connaître le `Gdk_Event_Type` de notre paramètre `Evenement`. Nous utiliserons pour cela une méthode de `Gdk_Event` :

Code : Ada

```
function Get_Event_Type (Event : Gdk_Event) return Gdk_Event_Type;
```

Voici ce que pourrait donner notre callback, revu et corrigé «para hablar Español» :

Code : Ada - P_Callbacks.adb

```
FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class;
    Evenement : GDK_Event; F : T_Fenetre) RETURN Boolean IS
    PRAGMA Unreferenced(Emetteur);
    Type_Evenement : CONSTANT Gdk_Event_Type := Get_Event_Type(Evenement);
BEGIN
```

```

IF Type_Evenement = Button_Press
  THEN CASE Get_Button(Evenement) IS
    WHEN 1 => F.Lbl.Set_Text("Hello world !");
    WHEN 3 => F.Lbl.Set_Text("Ouais, ouais... salut !")
  ;
  WHEN OTHERS => F.Lbl.Set_Text("");
END CASE;
ELSIF Type_Evenement = Gdk_2button_Press
  THEN F.Lbl.Set_Text("Ola ! Como esta ?") ; --Désolé pour les accents manquants.
END IF;
RETURN False;
END Dis_Bonjour;

```

Creusons un peu

Vous savez maintenant gérer les double-clics. Alors, heureux ? Je tiens toutefois à vous montrer quelque chose. Complétons le code précédent :

Code : Ada - P_Callbacks.adb

```

FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class;
  Evenement : GDK_Event;
  F          : T_Fenetre) RETURN Boolean IS
  PRAGMA Unreferenced(Emetteur);
  Type_Evenement : CONSTANT Gdk_Event_Type := Get_Event_Type(Evenement);
BEGIN
  IF Type_Evenement = Button_Press
  THEN put_line("clic");
    CASE Get_Button(Evenement) IS
      WHEN 1 => F.Lbl.Set_Text("Hello world !");
      WHEN 3 => F.Lbl.Set_Text("Ouais, ouais... salut !")
    ;
    WHEN OTHERS => F.Lbl.Set_Text("");
  END CASE;
  ELSIF Type_Evenement = Gdk_2button_Press
  THEN Put_Line("double clic");
    F.Lbl.Set_Text("Ola ! Como esta ?");
  END IF;
  RETURN False;
END Dis_Bonjour;

```

Compiler, exécutez votre programme et faites un double-clic sur le GTK_Button. Regardez maintenant la console. Elle affiche :

Code : Console

```
clic
clic
double clic
```

Autrement dit, le callback a été appelé 3 fois ! Soit une fois pour chaque clic puis une troisième fois pour le double clic. Ce n'est pas bien grave me direz-vous, mais cela doit vous amener à prendre quelques précautions : si vous souhaitez qu'un double-clic réalise une action, assurez-vous qu'un simple clic ne l'en empêche pas.

Et si on fouinait ?

Tant que nous avons le nez dans le package GDK.Event, profitons-en pour regarder quelques autres fonctions entourant Get_Event_Type() :

Code : Ada

```

function Get_Time   (Event : Gdk_Event) return Guint32;
  -- Renvoie l'heure à laquelle a eu lieu l'événement
function Get_X     (Event : Gdk_Event) return Gdouble;
function Get_Y     (Event : Gdk_Event) return Gdouble;
  -- Renvoient les coordonnées de la souris au moment de l'événement.
  -- Les coordonnées sont calculées par rapport à la fenêtre mère
function Get_X_Root (Event : Gdk_Event) return Gdouble;
function Get_Y_Root (Event : Gdk_Event) return Gdouble;
  -- Comme précédemment, sauf que les coordonnées sont calculées par rapport à la fenêtre dont est issu l'événement
function Get_Count (Event : Gdk_Event) return Gint;
  -- Renvoie le nombre d'événements en attente.

```

Le clavier

Reconnaître le clavier



Et maintenant : le clavier, le clavier, le clavier !!!

Vous êtes bien impatients ! Mais il est grand temps de nous en occuper. Nous allons faire appel aux GDK_Event_Type vus précédemment : Key_press et Key_release. Par exemple, si l'utilisateur clique sur une touche du clavier, le programme affichera un message en allemand :

Code : Ada - P_Callbacks.adb

```

FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class;
  Evenement : GDK_Event;
  F          : T_Fenetre) RETURN Boolean IS
  PRAGMA Unreferenced(Emetteur);
  Type_Evenement : CONSTANT Gdk_Event_Type := Get_Event_Type(Evenement);
BEGIN
  IF Type_Evenement = Button_Press
  THEN CASE Get_Button(Evenement) IS
    WHEN 1 => F.Lbl.Set_Text("Hello world !");
    WHEN 3 => F.Lbl.Set_Text("Ouais, ouais... salut !");
  ;
  WHEN OTHERS => F.Lbl.Set_Text("");
  END CASE;
  ELSIF Type_Evenement = Gdk_2button_Press
  THEN F.Lbl.Set_Text("Ola ! Como esta ?");
  ELSIF Type_Evenement = Key_Press
  THEN F.Lbl.Set_Text("Guten tag lieber Freund.");
  END IF;
  RETURN False;
END Dis_Bonjour;

```

Bien sûr, il est important que le GTK.Button soit sélectionné pour que cela arrive, qu'il ait le **focus**. Cela se traduit par un cadre en pointillé autour du texte du bouton. Notre programme ne contient qu'un seul bouton, donc le problème ne se pose pas mais pensez-y à l'avenir.

Reconnaître la lettre

Cherchons maintenant à connaître la touche utilisée. Le message sera en allemand seulement si l'utilisateur appuie sur une lettre minuscule. Nous allons utiliser la fonction get_string() qui renvoie le caractère obtenu par la touche du clavier mais sous forme de chaîne de caractère :

Code : Ada - P_Callbacks.adb

```

FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class;
  Evenement : GDK_Event;
  F          : T_Fenetre) RETURN Boolean IS

```

```

PRAGMA Unreferenced(Emetteur) ;
Type_Evenement : CONSTANT Gdk_Event_Type := 
Get_Event_Type(Evenement) ;
BEGIN
  IF Type_Evenement = Button_Press
    THEN CASE Get_Button(Evenement) IS
      WHEN 1 => F.Lbl.Set_Text("Hello world !");
      WHEN 3 => F.Lbl.Set_Text("Ouais, ouais ... salut
!") ;
      WHEN OTHERS => F.Lbl.Set_Text("") ;
    END CASE ;
  ELSIF Type_Evenement = Gdk_2button_Press
    THEN F.Lbl.Set_Text("Ola ! Como esta ?") ;
  ELSIF Type_Evenement = Key_Press
    THEN IF Get_String(Evenement)(1..2) = "z"
      THEN F.Lbl.Set_Text("Guten tag lieber
Freund.") ;
      ELSE F.Lbl.Set_Text("") ;
    END IF ;
  END IF ;
  RETURN False ;
END Dis_Bonjour;

```

Reconnaître la touche

Cela fonctionne mais un souci se pose : tous les caractères ne sont pas imprimables ! Si l'utilisateur appuie sur Tabulation ou la touche \leftarrow , le programme plante. Mieux vaudrait connaître le code de la touche activée que le caractère correspondant. Cela peut se faire avec la fonction `Get_Key_Val()` qui renvoie une valeur de type `Gdk_Key_Type`. Ce type est défini dans le package `Gdk.Types`, mais la plupart des constantes nécessaires se trouvent dans le package `Gdk.Types.Keyyms` que je vous conseille d'ouvrir à son tour. Vous y trouverez de nombreuses constantes. Par exemple, `GDK_Tab` (`Tab` (Tabulation), `GDK_Return` (`Entrée`), `GDK_Escape` (`Echap`), `GDK_Delete` (`Supprimer`), `GDK_Left`, `GDK_Right`, `GDK_Down`, `GDK_Up` (\leftarrow , \rightarrow , \downarrow et \uparrow), `GDK_Shift_L` ou `GDK_Shift_R` (Majuscule gauche et Majuscule droite)).

Les touches 'A' à 'Z' se nomment `GDK_A` à `GDK_Z`. Leurs valeurs, puisque ce ne sont que des nombres, sont comprises entre `16#0041#` et `16#005a#` (valeurs hexadécimales, vous l'aurez reconnu). Les minuscules, quant à elles se nomment `GDK_LC_a`, `GDK_LC_b...` (LC signifiant *lower case*). Leurs valeurs hexadécimales sont comprises entre `16#0061#` et `16#007a#`.

À nous de jouer maintenant ! Si l'utilisateur appuie sur une lettre minuscule, le programme lui parlera allemand, s'il appuie sur la tabulation, il lui parlera hollandais et sinon il lui parlera danois :

Code : Ada - P_Callbacks.adb

```

FUNCTION Dis_Bonjour(Emetteur : ACCESS GTK_Widget_Record'Class ;
                      Evenement : GDK_Event ;
                      F          : T_Fenêtre) RETURN Boolean IS
  PRAGMA Unreferenced(Emetteur) ;
  Type_Evenement : CONSTANT Gdk_Event_Type := 
Get_Event_Type(Evenement) ;
BEGIN
  IF Type_Evenement = Button_Press
    THEN CASE Get_Button(Evenement) IS
      WHEN 1 => F.Lbl.Set_Text("Hello world !");
      WHEN 3 => F.Lbl.Set_Text("Ouais, ouais ... salut
!") ;
      WHEN OTHERS => F.Lbl.Set_Text("") ;
    END CASE ;
  ELSIF Type_Evenement = Gdk_2button_Press
    THEN F.Lbl.Set_Text("Ola ! Como esta ?") ;
  ELSIF Type_Evenement = Key_Press
    THEN IF Get_Key_Val(Evenement) in 16#0061)..16#007a#
      THEN F.Lbl.Set_Text("Guten tag lieber Freund.") ;
    ELSE Get_Key_Val(Evenement) = GDK_Tab
      THEN F.Lbl.Set_Text("Goede dag lieve vriend.") ;
    ELSE F.Lbl.Set_Text("God dag kaere ven.") ;
    END IF ;
  END IF ;
  RETURN False ;
END Dis_Bonjour;

```

Bien sûr, pour que ce code fonctionne vous devrez penser à ajouter les lignes suivantes :

Code : Ada

```

WITH Gdk.Types ;           USE Gdk.Types;
WITH Gdk.Types.Keyyms ;   USE Gdk.Types.Keyyms ;

```

En résumé :

- Tout événement sur un widget, aussi anodin soit-il, déclenche l'émission d'un signal. C'est la capture de ce signal qui va permettre le déclenchement d'une procédure de rappel ou callback.
- Tout callback doit être connecté à un widget et un signal. Cela se fait grâce aux procédures `connect()` disponibles dans les sous-packages de `Gtk.Handlers`.
- Pour obtenir des interactions entre les différents widgets, vous pouvez utiliser le package `Gtk.Handlers.User_Callback`.
- Si vous ne trouvez pas votre bonheur dans les signaux proposés, cherchez parmi les signaux des widgets parents ou utilisez les `GDK_Event`.
- Faites appel aux types contrôlés et à vos connaissances en POO pour simplifier votre développement.

Les widgets I

Nous avons d'ores et déjà découvert de nombreux widgets : fenêtres, boutons, étiquettes ainsi que quatre conteneurs. Mais nous sommes loin d'avoir tout vu. Si je n'ai pas la prétention de vous faire découvrir toutes les possibilités de GTK, je ne peux tout de même pas vous abandonner avec si peu de widgets dans votre sac. Je vous propose de faire le tour des widgets suivants :

- Les étiquettes (nous allons approfondir ce que nous avons déjà vu)
- Les images
- Les zones de saisie : à une ligne, à plusieurs lignes, numériques
- De nouveaux boutons : à bascule, à cocher, fiens, radios
- Les séparateurs, les flèches, le calendrier et la barre de progression

Ce chapitre sera, j'en suis désolé, un véritable catalogue. Pour ne pas le rendre indigeste, j'ai tenté de l'organiser pour que vous puissiez facilement venir y piocher ce dont vous aurez besoin. Cette liste de widgets sera complétée ultérieurement par deux autres chapitres sur les widgets et par un second chapitre sur les conteneurs.

Les étiquettes

 On connaît déjà les étiquettes. Pourquoi en reparler ?

Nous en reparlons pour deux raisons :

- Nous n'avons vu que peu de méthodes applicables aux `GTK_Label`.
- Il est possible d'utiliser des caractères accentués dans vos `GTK_Label` mais si vous avez tenté, vous avez du remarquer la levée d'une exception. Il est également possible de mettre le texte en forme.

Fiche d'identité

- **Widget** : `GTK_Label`
- **Package** : `Gtk.Label`
- **Descendance** : `GTK_Widget >> GTK_Misc`
- **Description** : L'étiquette est une zone texte que l'utilisateur ne peut pas modifier. Cette zone peut être mise en forme à volonté.

Quelques méthodes des `GTK_Label`

Constructeurs

Pour l'instant, tout ce que nous avons vu à propos des étiquettes, c'est le constructeur `GTK_New()`. Vous aurez remarqué qu'il prend deux paramètres : le `GTK_Label`, bien évidemment, suivi d'un `string` correspondant à la phrase à afficher. Il est également possible de souligner un caractère avec la méthode `GTK_New_With_Mnemonic()` :

Code : Ada

```
GTK_New_With_Mnemonic(Lbl,"Cliquez sur _A.");
```

La ligne de code ci-dessus affichera le texte : « Cliquez sur A ». La règle est la même que pour les boutons. Voyons maintenant quelques-unes des méthodes de base des étiquettes :

Méthodes de base

Code : Ada

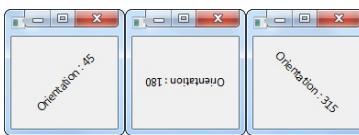
```
function Get_Label (Label : access Gtk_Label_Record) return
    UTF8_String;
procedure Set_Label (Label : access Gtk_Label_Record ;
    Str : UTF8_String);

function Get_Selectable (Label : access Gtk_Label_Record) return
    Boolean;
procedure Set_Selectable (Label : access Gtk_Label_Record;
    Setting : Boolean);

function Get_Use_Underline (Label : access Gtk_Label_Record) return
    Boolean;
procedure Set_Use_Underline (Label : access Gtk_Label_Record;
    Setting : Boolean);

function Get_Angle (Label : access Gtk_Label_Record) return
    Gdouble;
procedure Set_Angle (Label : access Gtk_Label_Record; Angle :
    Gdouble);
```

La méthode `Set_Label()` permet de redéfinir le contenu de l'étiquette, tandis que `Get_Label()` permet d'accéder à son contenu. L'ancien contenu est alors écrasé. `Set_Selectable()` autorise (ou non) l'utilisateur à sélectionner le texte du label. Par défaut, le texte n'est pas sélectionnable. La méthode `Set_Use_Underline()` permet, si cela n'a pas été fait, de souligner les lettres précédées d'un underscore. Enfin, la méthode `Set_Angle()` permet de définir l'orientation du texte de 0° à 360° (position normales). La mesure de l'angle est donnée en degrés et non en radians. (Voir la figure suivante)



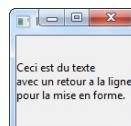
La méthode `Set_Angle`

Une première mise en forme

Il est possible d'afficher un texte sur plusieurs lignes. Pour cela, je vous conseille de vous créer une constante : « `newline : constant character := character'val(10) ;` ». Si votre texte doit comporter plusieurs lignes, il semblerait bon de déclarer préalablement votre texte dans une variable de type `string` pour plus de clarté, en utilisant le symbole de concaténation. Exemple (voir la figure suivante).

Code : Ada

```
Txt : string := "Ceci est du texte" & newline &
        "avec un retour à la ligne" & newline &
        "pour la mise en forme."
```



Le retour à la ligne

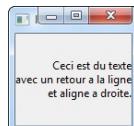
Dès lors que votre texte s'étend sur plusieurs lignes, il est possible de régler l'alignement du texte : aligné à gauche, à droite, centré ou justifié. Cela se fait à l'aide de la méthode suivante :

Code : Ada

```
procedure Set_Justify (Label : access Gtk_Label_Record;
    Jtype : Gtk.Enums.Gtk_Justification);
```

Le paramètre Jtype (dont le type `GTK_Justification` est, encore une fois, accessible via le package `Gtk.Enums`) peut prendre les valeurs suivantes :

- `Justify_Left` : le texte sera ligné à gauche
- `Justify_Right` : le texte sera ligné à droite (voir la figure suivante)
- `Justify_Center` : le texte sera centré
- `Justify_Fill` : le texte sera justifié, c'est-à-dire aligné à gauche et à droite.



Texte aligné à droite

Pour une mise en forme plus poussée

Les accents

Peut-être de voir plein d'erreurs levées dès que j'écris un 'é' ou un 'à' ! Il n'y aurait pas un moyen simple d'écrire normalement ?

Bien sûr que si. GTK gère de très nombreux caractères et notamment les caractères latins accentués. Pour cela, il utilise la norme Unicode et plus notamment l'encodage UTF-8. Pour faire simple (et réducteur, mais ce n'est pas le sujet de ce chapitre), il s'agit d'une façon de coder les caractères à la manière du code ASCII vu durant la partie IV. Si vous vous souvenez bien, le code ASCII utilisait 7 bits, les codes ASCII étendus utilisant soit 8 bits soit un octet. L'ASCII avait le gros inconvénient de ne proposer que des caractères anglo-saxons. Les ASCII étendus avaient eux l'inconvénient de ne pas être transposables d'un pays à l'autre (les caractères d'Europe de l'Ouest ne convenant pas à l'Europe orientale). Le codage UTF-8 utilise quant à lui jusqu'à 4 octets (soit 32 bits, ce qui fait 2^{32} caractères possibles) réglant ainsi la plupart de ces inconvénients et permettant même l'usage de caractères arabes, tibétains, arméniens...

Pour convertir proprement nos `String` en UTF-8, nous allons utiliser le package `Glib.Convert` et la fonction `Locale_To_UTF8()` qui convertit un `String` encodé dans le format "local" en `String` encodé en UTF-8 :

Code : Ada

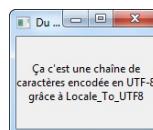
```
function Locale_To_UTF8(OS_String : String) return String;
```

Voici un exemple simplissime de code permettant l'emploi de caractères « bien de chez nous » (voir la figure suivante) :

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;        USE Gtk.Window ;
WITH Gtk.Label ;         USE Gtk.Label ;
WITH Glib.Convert ;      USE Glib.Convert ;
WITH Gtk.Enums ;         USE Gtk.Enums ;

PROCEDURE Texte IS
  Win    : Gtk.Window;
  Lbl   : Gtk.Label;
  newline : constant character := character'val(10);
  Txt : CONSTANT String := Locale_To_Utf8(
  "Ça c'est une chaîne de & Newline &
  <caractères encodés en UTF-8> & Newline &
  <grâce à Locale.To_UTF8>");
BEGIN
  Init ;
  Gtk_New(Win);
  Win.Set_Default_Size(100,100);
  win.set_title("Du texte !");
  Gtk_New(Lbl,txt);
  Lbl.Set_Justify(Justify_center);
  win.add(lbl);
  Win.show_all;
  Main;
END Texte;
```



Affichage des accents

La mise en forme avec Pango

Venons-en maintenant à la « vraie » mise en forme : gras, italique, souligné, barré, taille 5 ou 40, police personnalisée... car oui ! Tout cela est possible avec GTK ! Pour cela, une seule méthode sera utile :

Code : Ada

```
procedure Set_Use_Markup(Label : access Gtk_Label_Record ;
                          Setting : Boolean);
```

Il suffit que le paramètre `Setting` vaille `TRUE` pour pouvoir appliquer ces mises en forme. Reste maintenant à appliquer ces mises en forme. Pour cela, GTK fait appel à la bibliothèque Pango et à son langage à balise (Pango Markup Language).

Houlà ! Je ne comprenais déjà pas ce que voulait dire `Set_Use_Markup` mais là j'avoue que je me sens dépassé.

Le langage à balise de Pango, est un langage proche du HTML. Il consiste à placer des mots clés dans votre texte afin d'enclencher les zones bénéficiant d'une mise en forme particulière. Par exemple : `"Ceci est un texte avec BALISE DEBUT mise en forme BALISE FIN".` Les balises répondent toutefois à quelques règles. La balise ouvrante (celle du début) se note plus exactement entre deux chevrons : `<NOM_DE_BALISE>`. La balise fermante se note elle aussi entre deux chevrons mais est également précédée d'un slash `</NOM_DE_BALISE>`. L'exemple ci-dessus donnerait donc : `"Ceci est un texte avec <NOM_DE_BALISE> mise en forme </NOM_DE_BALISE>"`. Ainsi, les trois derniers mots bénéficieront des atouts apportés par la balise. Le principe du langage à balise est notamment utilisé en HTML ou XML.

Balises simples

Mais que dois-je écrire dans ces fameuses balises ?

Balise	Signification
<code></code>	Gras (bold)
<code><i></code>	Italique (italic)
<code><u></code>	souligné (underlined)
<code><s></code>	barré (striked)

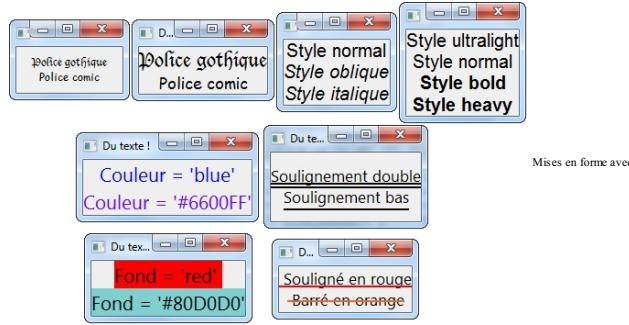
<sub>	en indice
<sup>	en exposant
<small>	petite taille
<big>	grande taille
<tt>	télétype

**Balise **

Une dernière balise existe, à savoir . Mais son utilisation est particulière. Seule, elle ne sert à rien, il faut y ajouter des attributs. Les attributs sont écrits à l'intérieur des chevrons de la balise ouvrante. Ils constituent des paramètres que l'on peut renseigner. Exemple : . Le nombre d'attributs n'est pas limité, vous pouvez tout autant n'en écrire qu'un seul qu'en écrire une dizaine. Vous remarquerez également que la valeur assignée à un attribut doit être écrite entre apostrophes.

Voici quelques-uns des attributs utilisables avec la balise :

Attribut	Valeurs possibles	Signification
font_family, face	'Times new Roman', 'Arial', 'Comic sans ms' ...	Indique la police utilisée
size	Valeurs absolues : 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' Valeurs relatives : 'smaller', 'larger' (plus petit et plus grand)	Indique la taille d'écriture
font_desc	Exemples : 'Arial 15', 'Comic sans ms 25' ...	Indique la police et la taille d'écriture
style	'normal', 'oblique', 'italic'	Indique l'obliquité du texte
weight	'ultralight', 'light', 'normal', 'bold', 'ultrabold', 'heavy' ou une valeur numérique	Indique le niveau de gras du texte
underline	'single', 'double', 'low', 'error'	Indique le type de soulignement (simple, double, bas ou ondulé façon faute d'orthographe)
foreground		Indique la couleur du texte
background		Indique la couleur de l'arrière plan
underline_color		Indique la couleur du soulignage
strikethrough_color		Indique la couleur du trait barrant le texte

**Les images**
Fiche d'identité

- Widget : GTK_Image
- Package : Gtk.Image
- Descendance : GTK_Widget >> GTK_Misc
- Description : Ce widget, comme son nom l'indique, affiche une image. Ce peut être un icône, un gif animé comme une photo de vacances au format JPEG.

Méthodes*Avec vos propres images*

Comme pour tous les widgets nous disposons d'une méthode GTK_New pour initialiser notre GTK_Image. Plus exactement, nous disposons de deux méthodes :

Code : Ada

```
procedure Gtk_New(Image : out Gtk_Image) ;
procedure Gtk_New(Image : out Gtk_Image;
                 Filename : UTF8_String) ;
```

La première ne fait que créer le widget sans y affecter de fichier image. Nous utiliserons de préférence la seconde méthode afin d'indiquer immédiatement l'adresse (relative ou absolue) du fichier image. Exemple avec l'image suivante :



Image pour le test

Code : Ada

```

WITH Gtk.Main ;      USE Gtk.Main ;
WITH Gtk.Window ;    USE Gtk.Window ;
WITH Gtk.Image ;     USE Gtk.Image ;

PROCEDURE Image IS
  Win   : Gtk.Window;
  Img   : Gtk.Image ;
BEGIN
  Init ;
  Gtk_New(Win) ;
  Win.Set_Default_Size(400,300) ;
  win.set_title("Une image ! ") ;
  Gtk_New(Img,"./picture.png") ;
  win.add(Img) ;
  Win.show_all ;
  Main ;
END Image ;

```



Fenêtre contenant une GTK_Image

Si vous optez pour la première méthode ou si vous souhaitez changer l'image, il vous faudra tout de même spécifier le chemin d'accès à l'image désirée grâce à la méthode `set()` :

Code : Ada

```
procedure Set(Image : access Gtk_Image_Record ; Filename : UTF8_String);
```

De même, si vous souhaitez modifier une image, il sera préférable de l'effacer auparavant grâce à la méthode `clear()` :

Code : Ada

```
procedure Clear(Image : access Gtk_Image_Record);
```

À partir d'icônes pré-stockés

Il existe plusieurs autres façons de charger une image : à partir d'une `Gdk_Pixbuf`, d'un `Gdk_Pixbuf_Record`, d'une `Gdk_Image`, d'un `GIcon`... vous n'étudierez la plupart de ces solutions que dans des cas bien précis. Mais il est une façon qui pourrait vous être utile. Gtk étant personnalisable, il dispose de thèmes graphiques ; lesquels thèmes comprennent également des thèmes d'icônes préenregistrés : les `GTK_Stock_Item`, disponibles via le package `Gtk STOCK`. Ce package fournit également tout un lot de chaînes de caractères constantes. Et cela tombe bien car `Gtk.Image` nous propose une autre méthode `Gtk_New` :

Code : Ada

```
procedure Gtk_New(Image : out Gtk.Image;
                  Stock_Id : UTF8_String;
                  Size     : Gtk.Enums.Gtk_Icon_Size);
```

Comme vous pouvez le constater, cette dernière attend comme paramètre une chaîne de caractères (`UTF8_String`) appelée `Stock_Id`. Cette méthode se chargera d'utiliser les sous-programmes fournis par le package `Gtk STOCK` afin de convertir cette chaîne de caractères en un `GTK_Stock_Item`. Ainsi, en écrivant :

Code : Ada

```
Gtk_New(Img,Stock_Cancel,...);
```

Vous obtiendrez l'icône suivante :



Icône Stock_Cancel

C'est bien gentil, mais tu n'as pas renseigné le dernier paramètre : `Size`. Je mets quoi comme taille, moi ?

Le type `Gtk.Icon_Size`, n'est rien d'autre qu'un nombre entier naturel. Mais si vous jetez un coup d'œil au package `Gtk.Enums` vous y découvrirez les tailles suivantes, classées de la plus petite à la plus grande :

- `Icon_Size_Invalid`: taille utilisée par Gtk pour indiquer qu'une taille n'est pas valide. Inutile pour vous.
- `Icon_Size_Menu`: taille utilisée pour les menus déroulants.
- `Icon_Size_Small_Toolbar`: taille utilisée pour les barres d'icônes.
- `Icon_Size_Large_Toolbar`: idem mais pour des barres utilisant de grands icônes.
- `Icon_Size_Button`: taille utilisée pour les boutons.
- `Icon_Size_Bnd`: taille utilisée pour le drag & drop, c'est-à-dire le glisser-déposer.
- `Icon_Size_Dialog`: taille utilisée pour les boîtes de dialogue.

Enfin, je vous livre les images de quelques icônes, mais je vous invite bien sûr à les essayer par vous-mêmes en lisant le package `Gtk STOCK`:



Exercice : créer un bouton avec icône

Pour égayer ce cours, voici un petit exercice : comment créer un bouton comportant à la fois du texte, mais également un icône comme celui-ci-dessous :



Un indice ? Rappelez-vous d'une petite phrase prononcée au début du chapitre sur les conteneurs. Je vous avais alors dit que les fenêtres et les boutons étaient des GTK.Container. Autrement dit, un bouton peut, tout autant qu'une boîte ou un alignement, contenir du texte, des images, d'autres conteneurs ou widgets. À vous de jouer.

[Secret \(cliquez pour afficher\)](#)

Code : Ada

```
WITH Gtk.Main ;      USE Gtk.Main ;
WITH Gtk.Window ;   USE Gtk.Window ;
WITH Gtk.Button ;   USE Gtk.Button ;
WITH Gtk.Image ;    USE Gtk.Image ;
WITH Gtk.Label ;    USE Gtk.Label ;
WITH Gtk.Box ;      USE Gtk.Box ;
WITH Gtk.Enums ;   USE Gtk.Enums ;
with gtk.fixed ;    use gtk.fixed ;

PROCEDURE Bouton_A_Image IS
  --On crée un type T_Button contenant tous les widgets nécessaires
  --Vous pouvez également en faire un type contrôlé
  TYPE T_Button IS RECORD
    Btn : Gtk_Button ;
    Box : Gtk_HBox ;
    Lbl : Gtk_Label ;
    Img : Gtk_Image ;
  END RECORD ;
  --Procédure d'initialisation du type T_Button
  PROCEDURE GTK_New(B : IN OUT T_Button ; Texte : String ;
  Adresse : String) IS
  BEGIN
    Gtk_New(B.Btn) ;
    Gtk_New_HBox(B.Box) ;
    Gtk_New(B.Lbl, Texte) ;
    Gtk_New(B.Img, Adresse) ;
    B.Box.Add(B.Box) ;
    B.Box.Pack_Start(B.Img) ;
    B.Box.Pack_Start(B.Lbl) ;
  END GTK_New ;
  B : T_Button ;
  F : Gtk_Fixed ;
  Win : Gtk_Window;
  BEGIN
    Init ;
    Gtk_New(Win) ;
    Win.Set_Default_Size(400,300) ;
    Win.Set_Title("Un bouton perso ! ") ;
    --on crée ici un conteneur de type GTK_Fixed pour mettre le bouton en évidence.
   Gtk_New(F) ;
    Win.Add(F) ;
    GTK_New(B,"Mon texte","./picture.png") ;
    F.Put(B.Btn,100,80) ;
    Win.Show_all ;
    Main ;
  END Bouton_A_Image ;
```

Les zones de saisie

La saisie de texte sur une ligne : les entrées

Fiche d'identité

- Widget : GTK_Entry ou GTK_GEntry (qui n'est qu'un sous-type).
- Package : Gtk.GEntry
- ! N'oubliez pas le G ! Ce package servant pour les GTK_Entry ainsi que pour d'autres types d'entrées, le G signifie général : General Entrées.
- Descendance : GTK_Widget >> GTK_Editable
- Description : Ce widget se présente sous la forme d'une zone de texte d'une seule ligne. Ce texte est tout à fait éditable par l'utilisateur et pourra servir, par exemple, à entrer un mot de passe. (Voir la figure suivante)



Méthodes

Je ne compte pas m'appesantir encore une fois sur le constructeur GTK_New(), celui-ci ne prend qu'un seul paramètre : votre widget. Vous serez donc capable de l'utiliser sans moi et je n'attirerai plus votre attention sur les constructeurs que lorsqu'ils nécessiteront des paramètres spécifiques.

Commençons donc par les méthodes concernant... le texte ! Comme les entrées sont faites pour cela, rien de plus logique :

Code : Ada

```
procedure Set_Text(The_Entry : access Gtk_Entry_Record;
                    Text : UTF8_String);
function Get_Text(The_Entry : access Gtk_Entry_Record) return
UTF8_String ;
function Get_Text_Length(The_Entry : access Gtk_Entry_Record)
return Uint16 ;
--METHODES OBSOLESSES

procedure Append_Text(The_Entry : access Gtk_Entry_Record ; Text :
UTF8_String);
procedure Prepend_Text(The_Entry : access Gtk_Entry_Record ; Text :
UTF8_String);
```

Tout d'abord, Set_Text() vous permet de fixer vous-même le texte de l'entrée ; utile pour définir un texte par défaut du genre *"tapez votre mot de passe ici"*. Puis, nous avons deux fonctions qui vous seront utiles dans vos callbacks : Get_Text() qui renvoie le texte tapé par l'utilisateur et Get_Text_Length() qui renvoie la longueur de la chaîne de caractères (utile pour indiquer à l'utilisateur que le mot de passe choisi est trop court par exemple). A noter que des méthodes obsolètes, mais toujours accessibles, pourraient vous faire gagner un peu de temps : Append_Text() pour ajouter des caractères à la fin du texte contenu dans l'entrée et Prepend_Text() pour l'ajouter au début (souvenez-vous les TAD, nous avions déjà utilisé les mots Append et Prepend). Ne vous inquiétez pas si le compilateur vous indique que ces méthodes sont obsolètes, elles fonctionnent tout de même. Venons-en maintenant à la taille de l'entrée :

Code : Ada

```
procedure Set_Max_Length(The_Entry : access Gtk_Entry_Record; Max :
Gint);
function Get_Max_Length(The_Entry : access Gtk_Entry_Record) return
Gint ;
procedure Set_Width_Chars(The_Entry : access Gtk_Entry_Record'Class;
Width : Gint);
function Get_Width_Chars(The_Entry : access Gtk_Entry_Record'Class)
return Gint ;
```

Les méthodes Set_Max_Length() et Get_Max_Length() permettent de définir ou de lire la longueur de l'entrée et donc

le nombre de caractères visibles. Si votre entrée a une taille de 8, elle ne pourra afficher que 8 caractères en même temps, ce qui n'empêchera pas l'utilisateur d'entrer un texte de 15 ou 20 caractères, l'affichage suivant le curseur. En revanche, si vous souhaitez limiter ou connaître le nombre de caractères inscriptibles, utilisez `Set_Width_Chars()` et `Get_Width_Chars()` (rappel : length signifie longueur, width signifie largeur). Passons maintenant à l'utilisation de l'entrée : celle-ci pouvant servir pour des mots de passe, il faudra pouvoir masquer (ou non) son contenu.

Code : Ada

```
procedure Set_Visibility(The_Entry : access Gtk_Entry_Record;
                           Visible : Boolean);
function Get_Visibility(The_Entry : access Gtk_Entry_Record) return
Boolean;

procedure Set_Invisible_Char (The_Entry : access Gtk_Entry_Record;
                             Char : Gunichar);
function Get_Invisible_Char (The_Entry : access Gtk_Entry_Record)
return Gunichar;
procedure Unset_Invisible_Char(The_Entry : access Gtk_Entry_Record);
```

Pour masquer le texte des ronds noirs (●●●), il faudra utiliser `Set_Visibility()` et que le paramètre `Visible` vaille `FALSE`. Il vous est possible de ne rien afficher du tout en utilisant `Set_Invisible_Char()` avec `char := 0`, ou de remplacer les ● par d'autre caractères. Pour ce faire, il faudra utiliser le package `glib.unicode` afin de convertir nos caractères en `Gunichar`:

Code : Ada

```
Mon_Entree.Set_Invisible_Char(UTF8_Get_Char("ø")); ;
```



`UTF8_Get_Char()` prend en paramètre un `string` pas un caractère, et renvoie un `Gunichar` qui, contrairement aux apparences, n'est rien d'autre qu'un entier 😊

Venons-en enfin à la mise en forme :

Code : Ada

```
procedure Set_Alignment (Ent : access Gtk_Entry_Record; Xalign :
Gfloat);
function Get_Alignment (Ent : access Gtk_Entry_Record) return
Gfloat;

function Get_Icon_Stock      (The_Entry : access Gtk_Entry_Record;
                             Icon_Pos : Gtk_Entry_Icon_Position)
return UTF8_String;
procedure Set_Icon_From_Stock(The_Entry : access Gtk_Entry_Record;
                             Icon_Pos : Gtk_Entry_Icon_Position;
                             Stock_Id : UTF8_String);
```

Les méthodes `Set_Alignment()` et `Get_Alignment()` permettent de définir ou de connaître la position du texte au sein de l'entrée. `Xalign` est un simple nombre en virgule flottante : quand il vaut `0.0`, le texte est aligné à gauche, quand il vaut `1.0` il est aligné à droite. Pour centrer le texte, il faudra donc que `Xalign` vaille `0.5`.

Avec `Set_Icon_From_Stock`, il est possible d'ajouter des icônes à votre entrée, comme nous le faisons avec nos images. Toutefois, vous devez préciser la position de l'icône. Deux valeurs sont possibles pour `Icon_Pos` :

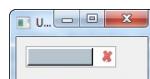
- `Gtk_Entry_Icon_Primary`: icône à gauche
- `Gtk_Entry_Icon_Secondary`: icône à droite



Une entrée avec texte centré et icône

Vous remarquerez que le curseur est centré par rapport à la place laissée disponible pour le texte.

Faire de votre entrée une barre de progression



Avec une barre de progression

Enfin, il est également possible de transformer votre `GTK_Entry` en une sorte de barre de progression :

Code : Ada

```
function Get_Progress_Fraction (The_Entry : access Gtk_Entry_Record)
return Gdouble;
procedure Set_Progress_Fraction(The_Entry : access Gtk_Entry_Record;
                                 Fraction : Gdouble);
function Get_Progress_Pulse_Step (The_Entry : access
Gtk_Entry_Record) return Gdouble;
procedure Set_Progress_Pulse_Step(The_Entry : access
Gtk_Entry_Record;
                                 Fraction : Gdouble);
procedure Progress_Pulse (The_Entry : access Gtk_Entry_Record);
```

La méthode `Set_Progress_Fraction()` permet de définir le pourcentage d'avancement de la barre de progression (`Fraction` vaudra `1.0` pour 100% d'avancement et `0.0` pour 0%).

Mais si vous souhaitez que vos callbacks fassent progresser la barre de manière régulière, vous devrez définir le pas, c'est à dire de quel pourcentage la barre progressera à chaque avancement. Pour définir ce pas, vous utiliserez `Set_Progress_Pulse_Step()`, puis il vous suffira d'utiliser `Progress_pulse()` pour faire faire progresser la barre de la valeur du pas que vous venez de définir.

Quelques signaux

Enfin, pour que vous puissiez utiliser toutes ces méthodes, voici quelques signaux qui vous seront utiles. Notez que vous pouvez les appeler soit via une chaîne de caractères, soit via le nom de la variable signal :

- `Signal_Activate / "activate"`: émis lorsque vous appuyez sur la touche Entrée.
- `Signal_Backspace / "backspace"`: émis lorsque vous appuyez sur la touche Retour Arrière.
- `Signal_Copy_Clipboard / "copy_clipboard"`: émis lorsque vous copiez du texte.
- `Signal_Cut_Clipboard / "cut_clipboard"`: émis lorsque vous coupez du texte.
- `Signal_Delete_From_Cursor / "delete_from_cursor"`: émis lorsque vous appuyez sur la touche Suppr.
- `Signal_Move_Cursor / "move_cursor"`: émis lorsque vous déplacez le curseur avec les touches fléchées.
- `Signal_Paste_Clipboard / "paste_clipboard"`: émis lorsque vous collez du texte.
- `Signal_Toggle_Overwrite / "toggle_overwrite"`: émis lorsque vous appuyez sur la touche Insér.

Saisie de texte multiligne



Ce widget sera revu plus tard pour parfaire son utilisation et sa présentation.

Fiche d'identité

- Widget : Gtk_Text_View.
- Package : Gtk.Text_View
- Descendance : GTK_Widget >> GTK_Container
- Description : Ce widget se présente sous la forme d'une page blanche sur laquelle l'utilisateur peut écrire tout ce qu'il souhaite.



Le widget Gtk_Text_View

Quelques méthodes

Je vais vous présenter maintenant quelques méthodes pour contrôler votre GTK_Text_View.

Code : Ada

```
procedure Set_Accepts_Tab(Text_View : access Gtk_Text_View_Record
                           ; Accepts_Tab : Boolean);
function Get_Accepts_Tab (Text_View : access Gtk_Text_View_Record) return Boolean;

procedure Set_Wrap_Mode(Text_View : access Gtk_Text_View_Record;
                        Wrap_Mode : Gtks.Enums.Gtk_Wrap_Mode);
function Get_Wrap_Mode (Text_View : access Gtk_Text_View_Record) return Gtks.Enums.Gtk_Wrap_Mode;

procedure Set_Cursor_Visible(Text_View : access Gtk_Text_View_Record;
                             Setting : Boolean := True);
function Get_Cursor_Visible (Text_View : access Gtk_Text_View_Record) return Boolean;
```

La méthode Set_Accepts_Tab() vous permettra de définir le comportement de votre widget lorsque l'utilisateur appuiera sur la touche de tabulation. Si le paramètre Accepts_Tab vaut **TRUE**, appuyer sur Tab créera une tabulation dans le GTK_Text_View. Si le paramètre Accepts_Tab vaut **FALSE**, appuyer sur la touche Tab permettra seulement de passer le focus d'un widget à un autre. La méthode Set_Wrap_Mode() indique quant à elle comment gérer les fins de ligne. Le paramètre Wrap_Mode peut valoir :

- Wrap_None : lorsque vous atteignez la fin de la ligne, le widget s'agrandit de manière à afficher les nouveaux caractères.
- Wrap_Char : lorsque vous atteignez la fin de la ligne, les nouveaux caractères sont écrits sur la ligne suivante, pouvant ainsi couper les mots en plein milieu.
- Wrap_Word : lorsque vous atteignez la fin de la ligne, le mot en cours d'écriture est placé sur la ligne suivante. En revanche, si le mot que vous écrivez est plus long que la ligne, le widget s'agrandira afin de loger le mot entier.
- Wrap_Word_Char : c'est le mode utilisé habituellement par les traitements de texte : en fin de ligne, le mot en cours d'écriture est placé tout entier sur la ligne suivante, à moins qu'il soit trop long, auquel cas il est coupé pour afficher les caractères suivants à la ligne.

Enfin, comme son nom l'indique, la méthode Set_Cursor_Visible() indique au widget si le curseur sera visible ou non. Venons-en maintenant à la mise en forme du widget.

Code : Ada

```
procedure Set_Justification(Text_View : access Gtk_Text_View_Record;
                            Justification : Gtks.Enums.Gtk_Justification);
function Get_Justification (Text_View : access Gtk_Text_View_Record) return Gtks.Enums.Gtk_Justification;

procedure Set_Left_Margin(Text_View : access Gtk_Text_View_Record;
                          Left_Margin : Gint);
function Get_Left_Margin (Text_View : access Gtk_Text_View_Record) return Gint;

procedure Set_Right_Margin(Text_View : access Gtk_Text_View_Record;
                           Right_Margin : Gint);
function Get_Right_Margin (Text_View : access Gtk_Text_View_Record) return Gint;

procedure Set_Indent(Text_View : access Gtk_Text_View_Record; Indent : Gint);
function Get_Indent (Text_View : access Gtk_Text_View_Record) return Gint;
```

Tout d'abord, la méthode Set_Justification() : celle-ci permet d'aligner le texte à gauche (**Justify_Left**), à droite(**Justify_Right**), au centre (**Justify_Center**) ou bien de le justifier, c'est à dire faire en sorte qu'il soit aussi bien aligné à gauche qu'à droite (**Justify_Fill**).

Les méthodes Set_Margin_Left() et Set_Margin_Right() permettent quant à elles de définir les marges du texte à gauche et à droite. Ainsi, l'instruction `Mon_Text_View.Set_Margin_Left(25)` réservera 25 pixels à gauche sur lesquels il sera impossible d'écrire.

Enfin, la méthode Set_Indent() permet de définir l'indentation du texte ou, pour laisser de côté le vocabulaire informatique, de définir la taille des alinéas. Vous savez, c'est une règle de typographie. Tout nouveau paragraphe commence par un léger retrait du texte de la première ligne (l'équivalent d'une tabulation très souvent). Très utile pour avoir des paragraphes distinguables et un texte clair. De même, nous avons les méthodes suivantes :

Code : Ada

```
procedure Set_Pixels_Above_Lines(Text_View : access Gtk_Text_View_Record;
                                   Pixels_Above_Lines : Gint);
function Get_Pixels_Above_Lines (Text_View : access Gtk_Text_View_Record) return Gint;

procedure Set_Pixels_Below_Lines(Text_View : access Gtk_Text_View_Record;
                                   Pixels_Below_Lines : Gint);
function Get_Pixels_Below_Lines (Text_View : access Gtk_Text_View_Record) return Gint;

procedure Set_Pixels_Inside_Wrap(Text_View : access Gtk_Text_View_Record;
                                   Pixels_Inside_Wrap : Gint);
function Get_Pixels_Inside_Wrap (Text_View : access Gtk_Text_View_Record) return Gint;
```

Les méthodes Set_Pixels_Above_Lines() et Set_Pixels_Below_Lines() permettent de définir le nombre de pixels au-dessus et au-dessous de chaque paragraphe. Pour bien comprendre le vocabulaire employé par GTK, vous devez avoir en tête que lorsque vous voyez un paragraphe composé de plusieurs lignes, pour GTK, il ne s'agit que d'une seule ligne d'une chaîne de caractère, c'est à dire d'un texte compris entre deux retours à la ligne. La méthode Set_Pixels_Inside_Wrap() définit quant à elle l'interligne au sein d'un même paragraphe. Enfin, voici une dernière méthode :

Code : Ada

```
procedure Set_Border_Window_Size(Text_View : access Gtk_Text_View_Record;
                                   The_Type : Gtks.Enums.Gtk_Text_Window_Type;
                                   Size      : Gint);
```

```
function Get_Border_Window_Size (Text_View : access
                                 Gtk_Text_View_Record;
                                 The_Type : Gtk.Enums.Gtk_Text_Window_Type) return Gint;
```

Elle permet de définir les bords de la zone de texte. Le paramètre `The_Type` peut valoir `Text_Window_Left` (bordure gauche), `Text_Window_Right` (bordure droite), `Text_Window_Top` (bordure haute) ou `Text_Window_Bottom` (bordure basse). Le paramètre `Size` correspond là encore au nombre de pixels.

Et pour quelques méthodes de plus

 Mais comment je fais si je veux écrire un texte justifié avec un titre centré ?

Eh bien le package `Gtk.Text_View` ne permet de manipuler que le widget `GTK_Text_View`, et pas d'affiner la présentation du texte. Si vous souhaitez entrer dans les détails, nous allons devoir utiliser les `Gtk_Text_Buffer`. Il ne s'agit plus d'un widget mais d'un `Object` (un objet défini par la bibliothèque `Glib`). Un buffer est une mémoire tampon dans laquelle sont enregistrées diverses informations comme le texte ou les styles et polices utilisés. Tout `GTK_Text_View` dispose d'un `Gtk_Text_Buffer`. Vous allez donc devoir :

1. Déclarer un `GTK_Buffer`
2. Déclarer un `GTK_Text_View`
3. Récupérer dans votre `GTK_Buffer`, le texte de votre `GTK_Text_View`

Code : Ada

```
...
Txt : GTK_Text_View ;
Tampon : GTK_Buffer ;
BEGIN
...
Tampon := Txt.Get_Buffer ;
```

Une fois l'adresse du texte récupérée dans votre buffer, vous allez pouvoir effectuer quelques recherches ou modifications. Par exemple, les méthodes `Get_Line_Count()` et `Get_Char_Count()` vous permettront de connaître le nombre de paragraphes et de caractères dans votre buffer :

Code : Ada

```
function Get_Line_Count(Buffer : access Gtk_Text_Buffer_Record)
return Gint;
function Get_Char_Count(Buffer : access Gtk_Text_Buffer_Record)
return Gint;
procedure Set_Text(Buffer : access Gtk_Text_Buffer_Record;
                    Text : UTF8_String);
procedure Insert_At_Cursor(Buffer : access Gtk_Text_Buffer_Record;
                            Text : UTF8_String);
```

La méthode `Set_Text()` vous permettra de modifier le texte. Attention toutefois, cela effacera tout le contenu précédent ! Si vous souhaitez seulement ajouter du texte à l'emplacement du curseur, utilisez plutôt `Insert_At_Cursor()`. Maintenant, nous allons voir comment obtenir du texte contenu dans le buffer ou insérer du texte à un endroit donné :

Code : Ada

```
function Get_Text (Buffer : access
                  Gtk_Text_Buffer_Record;
                  Start : Gtk.Text_Iter.Gtk_Text_Iter;
                  The_End : Gtk.Text_Iter.Gtk_Text_Iter;
                  Include_Hidden_Chars : Boolean := False) return
UTF8_String;
procedure Insert (Buffer : access Gtk_Text_Buffer_Record;
                  Iter : in out Gtk.Text_Iter.Gtk_Text_Iter;
                  Text : UTF8_String);
```

`Get_Text()` permettra d'obtenir le texte compris entre les paramètres `Start` et `The_End`, tandis que `Insert()` insérera du texte à l'emplacement `Iter`. Facile non ?

 Facile, facile... c'est quoi ces `Gtk_Text_Iter` ?

Ah oui, je vais un peu vite en besogne. Les `Gtk_Iter` sont des itérateurs, c'est-à-dire des emplacements dans le texte. Ces objets sont accessibles via le package `Gtk.Text_Iter`. Vous pouvez les contrôler de manière très précise en les déplaçant d'un ou plusieurs caractères. Cependant, je ne vois pour nous que quatre positions vraiment utiles :

- Le début du texte :

Code : Ada

```
procedure Get_Start_Iter(Buffer : access
                         Gtk_Text_Buffer_Record;
                         Iter : out
                         Gtk.Text_Iter.Gtk_Text_Iter);
```

- La fin du texte :

Code : Ada

```
procedure Get_End_Iter(Buffer : access Gtk_Text_Buffer_Record;
                       Iter : out
                       Gtk.Text_Iter.Gtk_Text_Iter);
```

- Le début et la fin d'une sélection :

Code : Ada

```
procedure Get_Selection_Bounds(Buffer : access
                               Gtk_Text_Buffer_Record;
                               Start : out
                               Gtk.Text_Iter.Gtk_Text_Iter;
                               The_End : out
                               Gtk.Text_Iter.Gtk_Text_Iter;
                               Result : out Boolean);
```

 Le paramètre `Result` indique si la sélection a une longueur non nulle ou pas. Si vous souhaitez savoir si du texte est sélectionné, il existe également la méthode « `FUNCTION Selection_Exists(Buffer : ACCESS
Gtk_Text_Buffer_Record) RETURN Boolean` ».

Voici un exemple d'utilisation :

Code : Ada

```
...
Txt : GTK_Text_View ; --Le widget GTK_Text_View
Tampon : GTK_Buffer ; --Son Buffer
Début,
Fin : Gtk_Iter ; --Les itérateurs de début et de fin
de sélection
Cm_Marche : Boolean ; --Un booléen pour savoir si la
sélection marche ou pas
BEGIN
...
GTK_New(Tampon) ;
```

```

Tampon := Txt.Get_Buffer ;
    --On effectue un affichage dans la console de l'intégralité
du texte
Tampon.Get_Start_Iter(Debut) ;
Tampon.Get_End_Iter(Fin) ;
Put_line("Le GTK_Text_View contient : ") ;
Put_line(Tampon.Get_Text(Debut,Fin)) ;

    --On effectue un affichage dans la console du texte
sélectionné
Tampon.Get_selection_Bounds(Debut,Fin,Ca_Marche) ;
Put_line("Vous avez sélectionné : ") ;
Put_line(Tampon.Get_Text(Debut,Fin)) ;
...

```

Venons-en maintenant à la mise en forme de notre texte. Malheureusement, la mise en forme du texte est plus complexe que pour les GTK_Entry. L'utilisateur ne va pas écrire de balise dans son texte, ce serait trop beau : un utilisateur qui serait programmeur en GTK_? Nous allons donc devoir créer des balises nous-même. Ces balises s'appellent des GTK_Text_Tag et leur usage est complexe. Vous devrez :

1. Créer votre GTK_Text_Tag grâce à GTK_New.
2. L'ajouter à la table des balises contenue dans le buffer en lui donnant un nom. Les tables de balises se nomment des GTK_Text_Tag_Table, mais nous ferons en sorte de ne pas y avoir recours.
3. Paramétriser votre Tag.

Le constructeur GTK_New() est disponible dans le package Gtk.Text_Tag et ne prend comme seul paramètre que votre Gtk_Text_Tag. Le premier soucis est d'ajouter votre tag à la table des tags du buffer. Pour faire cela, vous trouverez dans le package GtkText_Buffer la fonction suivante :

Code : Ada

```

function Create_Tag(Buffer : access Gtk_Text_Buffer_Record;
                     Tag_Name : String := "") return
    Gtk.Text_Tag.Gtk_Text_Tag;

```

Elle permet d'affecter votre tag à un tampon tout en lui attribuant un nom. Attention, la table des tags d'un buffer ne peut contenir deux tag portant le même nom. Nous verrons un exemple d'utilisation ci-dessous, rassurez-vous. Reste maintenant à paramétrer notre tag pour, par exemple, mettre du texte en gras. Pour cela, vous aurez besoin du package Pango.Enums (pas de GTK !). Celui-ci comporte toutes sortes de types énumérés correspondant à différents styles mais surtout, il implémente le package générique Glib.Generic_Properties. Vous me suivez ? Plus simplement, c'est comme si notre package Pango.Enums disposait de la méthode suivante pour définir les propriétés d'un GTK_Text_Tag (je simplifie pour y voir clair) :

Code : Ada

```

procedure Set_Property(Object : access GTK_Text_Tag_Record'Class;
                       Name   : Property ;
                       Value  : Property_Type);

```

Le paramètre Name est le nom d'une propriété, cela permet d'indiquer ce que vous souhaitez modifier (gras, italique, barré...). Vous trouverez les valeurs possibles dans le package GTK_Text_Tag. Par exemple, la propriété Gras se nomme Weight_Property. Le paramètre Value permet de préciser la propriété (gras léger, gras épais...) et vous trouverez les valeurs possibles dans Pango.Enums :

Code : Ada

```

type Weight is
  (Pango_Weight_Ultralight,
   Pango_Weight_Light,
   Pango_Weight_Normal,
   Pango_Weight_Medium,
   Pango_Weight_Semi_Bold,
   Pango_Weight_Bold,
   Pango_Weight_Ultrabold,
   Pango_Weight_Heavy);

```

  Excuse-moi, mais j'ai beau te relire, j'ai vraiment du mal à comprendre. Je me perds dans les packages et les méthodes.

Résumons : dans GTK_Text_Tag nous trouverons le type GTK_Text_Tag, son constructeur GTK_New() ainsi que des noms de propriétés (Style_Property, Underline_Property, Strikethrough_Property, Weight_Property...). Dans Pango.Enums nous trouverons plus de précisions pour différentes propriétés (type de gras, d'italique...) ainsi que la méthode Set_Property() (implicitement). La méthode Create_Tag() permettant d'ajouter le tag à la table des tags est disponible dans GTK_Text_Buffer. Un exemple résumerait tout cela encore mieux :

Code : Ada

```

...
tag : GTK_Text_Tag ;
tampon : GTK_Text_Buffer ;
BEGIN
...
    --création du tag
    gtk_new(tag) ;
    --définition du tag : propriété gras et plus exactement
    UltraGras ;
    Set_Property(Tag, Weight_Property, Pango_Weight_Ultrabold) ;
    --ajout du tag dans la table du buffer sous le nom "gras"
    Tag := Tampon.Create_Tag("gras") ;

```

Maintenant, il ne vous restera plus qu'à insérer du texte déjà formaté à l'aide de votre tag ou simplement d'appliquer votre tag à une partie du buffer à l'aide des méthodes suivantes :

Code : Ada

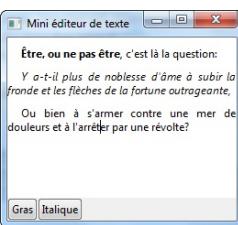
```

--INSERER DU TEXTE FORMATE
procedure Insert_With_Tags          --À l'aide de
votre variable Tag
  (Buffer : access Gtk_Text_Buffer_Record;
   Iter   : in out Gtk_Text_Iter.Gtk_Text_Iter;
   Text   : UTF8_String;
   Tag    : Gtk.Text_Tag.Gtk_Text_Tag);
procedure Insert_With_Tags_By_Name   --À l'aide du nom
de votre Tag : "gras"
  (Buffer : access Gtk_Text_Buffer_Record;
   Iter   : access Gtk.Text_Tag.Gtk_Text_Tag_Record'Class;
   Start  : Gtk.Text_Iter.Gtk_Text_Iter;
   The_End: Gtk.Text_Iter.Gtk_Text_Iter);
--APPLIQUER UN FORMAT
procedure Apply_Tag                --À l'aide de
votre variable Tag
  (Buffer : access Gtk_Text_Buffer_Record;
   Tag   : access Gtk.Text_Tag.Gtk_Text_Tag_Record'Class;
   Start  : Gtk.Text_Iter.Gtk_Text_Iter;
   The_End: Gtk.Text_Iter.Gtk_Text_Iter);
procedure Apply_Tag_By_Name        --À l'aide du nom
de votre Tag : "gras"
  (Buffer : access Gtk_Text_Buffer_Record;
   Name   : String;
   Start  : Gtk.Text_Iter.Gtk_Text_Iter;
   The_End: Gtk.Text_Iter.Gtk_Text_Iter);

```

Un exemple (ou un exercice)

Voici un exemple de code utilisant les GTK_Text_View, avec les buffers et les tags. Il s'agit d'un mini éditeur de texte : il suffit de sélectionner du texte puis de cliquer sur les boutons Gras ou Italique pour appliquer la mise en forme.



Code : Ada

```

WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Text_View ; USE Gtk.Text_View ;
WITH Gtk.Enums ; USE Gtk.Enums ;
WITH Gtk.Handlers ; USE Gtk.Handlers ;
WITH Gtk.Box ; USE Gtk.Box ;
WITH Gtk.Button ; USE Gtk.Button ;
WITH Glib.Convert ; USE Glib.Convert ;
WITH Gtk.Text_Buffer ; USE Gtk.Text_Buffer ;
WITH Gtk.Text_Iter ; USE Gtk.Text_Iter ;
WITH Gtk.Text_Tag ; USE Gtk.Text_Tag ;
WITH Pango.Enums ; USE Pango.Enums ;

PROCEDURE Texte IS
  TYPE T_Fenetre IS RECORD
    Win      : Gtk.Window;
    Txt     : Gtk.Text_View;
    Btn_Gras, Btn_Ital : Gtk_Button;
    VBox   : GTK_VBox;
    HBox   : GTK_HBox;
  END RECORD;
  --INITIALISATION D'UN OBJET T_FENETRE

PROCEDURE Init(F : IN OUT T_Fenetre) IS
  Tag      : Gtk.Text_Tag;
  Tampon  : GTK_Text_Buffer;
BEGIN
  Init;
  --Création de la fenêtre
  Gtk_New(F.Win);
  F.Win.Set_Default_Size(250,200);
  F.Win.Set_Label(Gtk.To_Utf8("Mini éditeur de texte"));
  --Création des boutons
  Gtk_New(F.Btn_Gras,"Gras");
  Gtk_New(F.Btn_Ital,"Italique");
  --Création du GTK_Text_View
  Gtk_New(F.Txt);
  F.Txt.Set_Wrap_Mode(Wrap_Word_char);
  F.Txt.Set_Justification(Justify_Fill);
  F.Txt.Set_Pixels_Above_Lines(10);
  F.Txt.Set_Indent(15);
  --Création des boîtes et organisation de la fenêtre
  GTK_New_VBox(F.VBox);
  GTK_New_HBox(F.HBox);
  F.VBox.Pack_Start(F.Txt);
  F.VBox.Pack_Start(F.Btn_Gras, Expand=> False, Fill => False);
  F.HBox.Pack_Start(F.Btn_Ital, Expand=> False, Fill => False);
  F.HBox.Pack_Start(F.Btn_Gras, Expand=> False, Fill => False);
  F.VBox.Pack_Start(F.Btn_Ital, expand=> false, fill => false);
  F.Win.Add(F.VBox);

  F.Win.Show_All;
  --Création des tags et ajout à la table du buffer
  gtk_new_tag;
  Tampon := F.Txt.Get_Buffer;
  Tag := Tampon.Create_Tag("gras");
  Set_Property(Tag, Weight_Property, Pango_Weight_Ultrabold);
  Set_Property(Tag, Style_Property, Pango_Style_Italic);
  -- Tag := Tampon.Create_Tag("italique");
  set_property(Tag, Style_Property, Pango_Style_Italic);
END Init;

--LES CALLBACKS
PACKAGE Callback IS
  NEW
  Gtk.Handlers.User_Callback(GTK_Button_Record, T_Fenetre);
  USE Callback;

  PROCEDURE To_Bold(Emetteur : ACCESS GTK_Button_Record'Class; F : T_Fenetre) IS
    PRAGMA Unreferenced(Emetteur);
    Tampon : GTK_Text_Buffer;
    Debut,Fin : GTK_Text_Iter;
    Resultat : Boolean;
  BEGIN
    Gtk_New(Tampon);
    Tampon := F.Txt.Get_Buffer;
    Tampon.Get_Selection_Bounds(debut,fin,resultat);
    Initialisation_des_Iter;
    Tampon.apply_tag_by_name("gras",debut,fin);
    Application_du_format_Gras;
  END To_Bold;

  PROCEDURE To_Italic(Emetteur : ACCESS GTK_Button_Record'Class; F : T_Fenetre) IS
    PRAGMA Unreferenced(Emetteur);
    Tampon : GTK_Text_Buffer;
    Debut,Fin : GTK_Text_Iter;
    Resultat : Boolean;
  BEGIN
    Gtk_New(Tampon);
    Tampon := F.Txt.Get_Buffer;
    Tampon.Get_Selection_Bounds(debut,fin,resultat);
    Initialisation_des_Iter;
    Tampon.apply_tag_by_name("italique",debut,fin);
    Application_du_format_Italique;
  END To_Italic;

  F : T_Fenetre;
  BEGIN
    Init(F);
    Connect(F.Btn_Gras,"clicked",To_Bold'ACCESS, F);
    connect(F.Btn_Ital,"clicked",To_Italic'access, F);
    Main;
  END Texte;

```

Quelques signaux

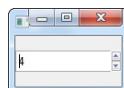
Les GTK_Text_View aussi peuvent émettre des signaux. Vous retrouverez la plupart des signaux vus pour les GTK_Entry :

- Signal_Activate / "activate"
- Signal_Backspace / "backspace"
- Signal_Copy_Clipboard / "copy_clipboard"
- Signal_Cut_Clipboard / "cut_clipboard"
- Signal_Delete_From_Cursor / "delete_from_cursor"
- Signal_Move_Cursor / "move_cursor"
- Signal_Paste_Clipboard / "paste_clipboard"
- Signal_Toggle_Overwrite / "toggle_overwrite"

Saisie numérique

Fiche d'identité

- **Widget :** Gtk_Spin_Button
- **Package :** Gtk.Spin_Button
- **Descendance :** GTK_Widget >> GTK_Editable >> GTK_GEntry
- **Description :** Ce widget présente une zone d'affichage d'un nombre, accompagnée de deux petits boutons pour augmenter ou diminuer ce nombre.



La saisie numérique

Méthodes

Le constructeur est cette fois plus exigeant :

Code : Ada

```
procedure Gtk_New(Spin_Button : out Gtk_Spin_Button;
                 Min        : Gdouble;
                 Max        : Gdouble;
                 Step       : Gdouble);
procedure Set_Increments(Spin_Button : access
                         Gtk_Spin_Button_Record;
                         Step      : Gdouble;
                         Page     : Gdouble);
procedure Get_Increments(Spin_Button : access
                         Gtk_Spin_Button_Record;
                         Step      : out Gdouble;
                         Page     : out Gdouble);
procedure Set_Range(Spin_Button : access Gtk_Spin_Button_Record;
                    Min        : Gdouble;
                    Max        : Gdouble);
procedure Get_Range(Spin_Button : access Gtk_Spin_Button_Record;
                    Min        : out Gdouble;
                    Max        : out Gdouble);
```

Vous devrez lui indiquer la valeur minimale, la valeur maximale ainsi que le pas (Step), c'est-à-dire la valeur de l'augmentation ou de la diminution qui aura lieu à chaque fois que l'utilisateur appuiera sur les boutons correspondants. Attention, ces valeurs sont des GDouble, c'est-à-dire des nombres flottants double précision. Rassurez-vous, si vous souhaitez un compteur entier, GTK n'affichera pas de «.0» à la fin de vos nombres. De plus, vous pouvez modifier ces valeurs grâce aux méthodes Set_Increments() et Set_Range(). Si Set_Range() se contente de redéfinir le minimum et le maximum, la méthode Set_Increments() permet de redéfinir deux types de pas : le «pas simple» (Step) et le «pas large» (Page). Le pas large correspond à l'augmentation obtenue lorsque vous appuyerez sur les touches Page Haut ou Page Bas de votre clavier (les deux touches à côté de Fin et Début) ou lorsque vous cliquerez sur les boutons d'augmentation/diminution avec le bouton n°2 de la souris (la molette centrale pour une souris trois boutons). Si vous cliquez à l'aide du bouton n°3 (bouton droit) vous attendez la plus petite ou plus grande valeur.

Code : Ada

```
procedure Set_Digits(Spin_Button : access Gtk_Spin_Button_Record;
                      The_Digits : Uint);
function Get_Digits(Spin_Button : access Gtk_Spin_Button_Record)
return Uint;
procedure Set_Value(Spin_Button : access Gtk_Spin_Button_Record;
                   Value     : Gdouble);
function Get_Value(Spin_Button : access Gtk_Spin_Button_Record)
return Gdouble;
```

Vous pouvez également préciser à l'aide de Set_Digits() le nombre de chiffres que vous désirez après la virgule. Enfin, pour vos callbacks, vous pourrez utiliser les méthodes Set_Value() et Get_Value() qui vous permettront de définir ou d'obtenir la valeur du GTK_Spin_Button.



Comme pour les autres widgets, je ne parle pas de l'intégralité des méthodes disponibles mais seulement des plus importantes. N'hésitez pas à fouiller dans les packages pour y découvrir des méthodes qui pourraient vous manquer.

Signal

Un signal devrait vous servir ici. Il s'agit du signal Signal_Value_Changed / "value_changed" qui est émis dès lors que la valeur du GTK_Spin_Button est modifiée (soit en cliquant sur les boutons d'augmentation/réduction soit en tapant la valeur manuellement et en appuyant sur Entrée).

Saisie numérique par curseur

Fiche d'identité

- **Widget :** Gtk_Scale, Gtk_HScale et Gtk_VScale.
- **Package :** Gtk.Scale
- **Descendance :** GTK_Widget >> GTK_Range
- **Description :** Ce widget permet de saisir un nombre en déplaçant un curseur.



Curseur GTK_Hscale

Méthodes

Les curseurs (appelés échelles par GTK) ont deux orientations possibles : à la verticale ou à l'horizontale. Il existe donc un constructeur pour chacun d'eux (en fait il y a même deux pour chacun, mais nous n'en verrons qu'un seul) :

Code : Ada

```
procedure Gtk_New_Hscale(Scale : out Gtk_Hscale;
                        Min   : Gdouble;
                        Max   : Gdouble;
                        Step  : Gdouble);
procedure Gtk_New_Vscale(Scale : out Gtk_Vscale;
                        Min   : Gdouble;
                        Max   : Gdouble;
                        Step  : Gdouble);

function Get_Orientation(Self : access Gtk_Scale_Record) return
Gtk.Enums.Gtk_Orientation;
procedure Set_Orientation(Self   : access Gtk_Scale_Record;
                         Orientation : Gtk.Enums.Gtk_Orientation);
```

Je ne reviens pas sur les paramètres Min, Max et Step, ce sont les mêmes que pour les GTK_Spin_Button. Si jamais vous voulez modifier ou connaître l'orientation de votre curseur, vous pouvez utiliser Get_Orientation() et Set_Orientation(). Deux types d'orientations sont disponibles :Orientation_Horizontal et Orientation_Vertical. Ai-je besoin de traduire ? 😊

Code : Ada

```

function Get_Digits (Scale : access Gtk_Scale_Record) return Gint;
procedure Set_Digits(Scale : access Gtk_Scale_Record;
                      Number_Of_Digits : Gint);

function Get_Draw_Value (Scale : access Gtk_Scale_Record) return
Boolean;
procedure Set_Draw_Value(Scale : access Gtk_Scale_Record;
                        Draw_Value : Boolean);

procedure Add_Mark (Scale : access Gtk_Scale_Record;
                    Value : Gdouble;
                    Position : Gtk.Enums.Gtk_Position_Type;
                    Markup : UTF8_String);
procedure Clear_Marks(Scale : access Gtk_Scale_Record);

function Get_Value_Pos (Scale : access Gtk_Scale_Record) return
Gtk.Enums.GTK_Position_Type;
procedure Set_Value_Pos(Scale : access Gtk_Scale_Record;
                      Pos : Gtk.Enums.Gtk_Position_Type);

```

Comme précédemment, pour connaître ou modifier le nombre de chiffres après la virgule à afficher (Number_Of_Digits), utilisez les méthodes Get_Digits() et Set_Digits(). Attention, cette dernière peut modifier votre pas, s'il n'est pas suffisamment précis. Vous pouvez ne pas afficher la valeur correspondant à votre curseur en utilisant la méthode Mon_Gtk_Scale.Set_Draw_Value(FALSE).

Autres méthodes : Add_mark() qui ajoute une marque à votre curseur et Clear_marks() qui les effacent toutes. Qu'est-ce qu'une marque me direz-vous ? Eh bien, c'est simplement un petit trait sur le bord de l'échelle permettant d'indiquer une position précise (le milieu, une ancienne valeur...) comme sur l'exemple ci-dessous :



Curseur avec marque

Le paramètre Value indique simplement le lieu où doit se trouver la marque. Le paramètre Position indique s'il doit se situer au-dessus ou en-dessous (Pos_Top ou Pos_Bottom, valable seulement pour les GTK_HScale), à gauche ou à droite (Pos_Left ou Pos_Right, valable seulement pour les GTK_VScale). Enfin, Markup est le texte qui sera inscrit à côté de la marque.

Dans le même esprit, Get_Value_Pos() et Set_Value_Pos() permettent de connaître ou de définir la position de la valeur du curseur : est-elle affichée au-dessus, au-dessous, à gauche ou à droite du curseur ?

Code : Ada

```

function Get_Value (The_Range : access Gtk_Range_Record) return
Gdouble;
procedure Set_Value(The_Range : access Gtk_Range_Record;
                     Value : Gdouble);
procedure Set_Increments(The_Range : access Gtk_Range_Record;
                         Step : Gdouble;
                         Page : Gdouble);
procedure Set_Range(The_Range : access Gtk_Range_Record;
                   Min : Gdouble;
                   Max : Gdouble);

```

Par héritage, les GTK_Scale bénéficient également des méthodes ci-dessus. Elles jouent le même rôle que pour les GTK_Spin_Button, je ne vais donc pas vous les réexpliquer.

D'autres boutons

Vous venez de terminer la partie la plus complexe de ce chapitre. Nous allons maintenant nous détendre en découvrant quatre nouveaux boutons plus spécifiques : les boutons à bascule, les boutons-liens, les boutons à cocher et les boutons radios.

Boutons à bascule

Fiche d'identité

- Widget : GTK_Toggle_Button.
- Package : Gtk.Toggle_Button
- Descendance : GTK_Widget >> GTK.Container >> GTK_Bin >> GTK_Button
- Description : Il s'agit d'un bouton d'apparence classique, mais qui ne peut en réalité être que dans l'un des deux états suivants : actif(enfoncé) ou inactif(relevé).



Boutons à bascule

Méthodes

Les Toggle_Button se créent comme des boutons classiques avec GTK_New() ou Gtk_New_With_Mnemonic(). La nouveauté est de pouvoir activer ou non le bouton à bascule :

Code : Ada

```

function Get_Active(Toggle_Button : access
                     Gtk_Toggle_Button_Record) return Boolean;
procedure Set_Active(Toggle_Button : access
                     Gtk_Toggle_Button_Record;
                     Is_Active : Boolean);

```

Pour que le bouton soit activé, il suffit que Is_Active vaille TRUE. Enfin, il est également possible de « bloquer » le bouton si par exemple des paramètres renseignés par l'utilisateur étaient incompatibles avec son activation (dans un jeu, un bouton « partie rapide » serait bloqué si le joueur demandait à jouer contre 30 adversaires). Pour cela il suffit d'utiliser les méthodes suivantes avec Setting à TRUE :

Code : Ada

```

function Get_Inconsistent(Toggle_Button : access
                           Gtk_Toggle_Button_Record) return Boolean;
procedure Set_Inconsistent(Toggle_Button : access
                           Gtk_Toggle_Button_Record;
                           Setting : Boolean := True);

```

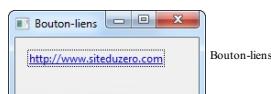
Signaux

Les Toggle_Button apportent un nouveau signal : Signal_Toggled ou "toggled" qui signifie que le bouton a basculé d'un état à un autre (actif vers inactif ou réciproquement).

Boutons-liens

Fiche d'identité

- Widget : GTK_Link_Button.
- Package : Gtk.Link_Button
- Descendance : GTK_Widget >> GTK.Container >> GTK_Bin >> GTK_Button
- Description : Ce bouton se présente comme un lien internet, gardant en mémoire s'il a été cliqué ou non.

**Méthodes****Code : Ada**

```
procedure Gtk_New (Widget : out Gtk_Link_Button;
                  Uri : String);
procedure Gtk_New_With_Label (Widget : out Gtk_Link_Button;
                             Uri : String;
                             Label : String);
```

Lors de la création d'un `Gtk_Link_Button`, vous devrez renseigner l'URI vers laquelle il est sensé pointé, c'est à dire l'adresse internet ou le fichier visé. Exemple : "<http://siteduzero.com>". Si vous souhaitez que le bouton affiche un texte spécifique plutôt que l'adresse du fichier ou de la page cible, utilisez la méthode `Gtk_New_With_Label()`.

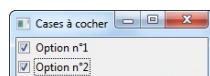
Code : Ada

```
function Get.Uri      (Link_Button : access Gtk_Link_Button_Record)
return String;
procedure Set.Uri    (Link_Button : access Gtk_Link_Button_Record;
                     Uri       : String);
function Get.Visited  (Link_Button : access Gtk_Link_Button_Record)
return Boolean;
procedure Set.Visited (Link_Button : access Gtk_Link_Button_Record;
                     Visited   : Boolean);
```

Avec les méthodes précédentes vous pourrez obtenir ou modifier l'adresse URI du bouton. Vous pourrez également savoir si le lien a été visité ou le réinitialiser. Enfin, les `Gtk_Link_Button` ne présentent pas de signaux particuliers (pas davantage que les boutons classiques).

Boutons à cocher**Fiche d'identité**

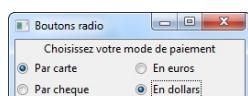
- **Widget :** `GTK_Check_Button`.
- **Package :** `GtkCheck_Button`
- **Descendance :** `GTK_Widget >> GTK.Container >> GTK_Bin >> GTK_Button >> GTK_Toggle_Button`
- **Description :** Il s'agit d'un widget pouvant être coché ou décoché à volonté. Il est généralement utilisé pour paramétriser les options.

**Méthodes et signaux**

L'utilisation des `GTK_Check_Button` est simplissime : ce sont des `GTK_Toggle_Button`, ils ont donc les mêmes méthodes et signaux.

Boutons radios**Fiche d'identité**

- **Widget :** `Gtk_Radio_Button`.
- **Package :** `Gtk_Radio_Button`
- **Descendance :** `GTK_Widget >> GTK.Container >> GTK_Bin >> GTK_Button >> GTK_Toggle_Button >> GTK_Check_Button`.
- **Description :** Ce widget est généralement utilisé en groupe, pour choisir entre diverses options s'excluant les unes les autres.



Deux groupes de boutons radio : le moyen de paiement et le type de monnaie. Mais dans chaque groupe, un seul bouton peut être coché à la fois.

Méthodes

Lorsque vous créez un `Gtk_Radio_Button`, vous créez également une `GSList` (plus exactement une `Gtk.Widget.Widget_SList.GSList`). Cette liste permet à votre programme de lier plusieurs boutons radios entre eux : lorsque l'utilisateur clique sur l'un d'entre eux, cela décochera tous les autres de la liste. Heureusement, GTKAda nous simplifie la vie. Pour ne pas créer de `GSList` en plus de nos `Gtk_Radio_Button`, il existe un constructeur liant les `Gtk_Radio_Button` entre eux :

Code : Ada

```
procedure Gtk_New (Radio_Button : out Gtk_Radio_Button;
                  Group      : Gtk_Radio_Button;
                  Label      : UTF8_String := "");
```

Le paramètre `Radio_Button` est bien sûr le bouton radio que vous souhaitez créer. Le paramètre `Group` est l'un des boutons radio de la liste et `Label` est bien entendu l'étiquette du bouton. Voici un exemple d'utilisation des boutons radio ; il s'agit de proposer à l'utilisateur de payer par carte ou par chèque, en euros ou en dollars :

Code : Ada

```
...
  BtnA1, BtnA2  : GTK_Radio_Button ;
  BtnB1, BtnB2  : GTK_Radio_Button ;
BEGIN
  --Première liste de boutons
  GTK_New(BtnA1,NULL, "Par carte") ;
  GTK_New(BtnA2, BtnA1, "Par cheque") ;

  --Deuxième liste de boutons
  GTK_New(BtnB1,NULL, "En euros") ;
  GTK_New(BtnB2, BtnB1, "En dollars") ;
```

Vous aurez remarqué que pour le premier bouton de chaque liste, il suffit d'indiquer `NULL` pour qu'il ne soit lié à aucun autre bouton (bah oui, les `GTK_Radio_Button` sont des pointeurs sur des `GTK_Radio_Button_Record`, vous vous souvenez).

Enfin, ne cherchez pas davantage de méthodes : les boutons radio sont des boutons à cocher, qui eux-mêmes sont des boutons à bascule, qui eux-mêmes sont des boutons... Ouf ! Les histoires de famille c'est compliqué, mais l'avantage c'est qu'il n'y aura pas de nouvelles méthodes ou de nouveaux signaux à apprendre.

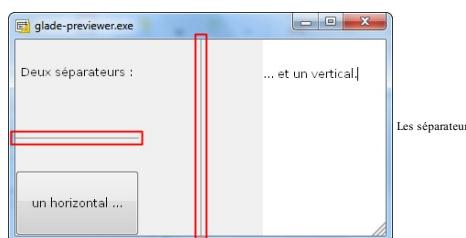
Widgets divers

Pour clore ce chapitre, un peu de légèreté avec des widgets anodins et simples d'utilisation.

Les séparateurs

Fiche d'identité

- **Widget**: Gtk_Separator, Gtk_VSeparator, Gtk_HSeparator.
- **Package** : Gtk.Separator
- **Descendance** : GTK_Widget
- **Description** : Ce widget est une simple ligne séparant différentes zones d'une même fenêtre.



Méthodes

Il existe deux sortes de séparateurs : les séparateurs horizontaux (GTK_HSeparator_Record) et les séparateurs verticaux (GTK_VSeparator_Record). Ce qui implique donc deux constructeurs d'une simplicité enfantine :

Code : Ada

```
procedure Gtk_New_Vseparator (Separator : out Gtk_Vseparator);
procedure Gtk_New_Hseparator (Separator : out Gtk_Hseparator);
```

Mais comme il ne s'agit que de deux sous-types finalement très semblables, on peut aisément transformé un GTK_HSeparator en GTK_VSeparator. Il suffit de modifier leur orientation avec :

Code : Ada

```
--Pour connaître l'orientation du séparateur
function Get_Orientation(Self : access Gtk_Separator_Record) return
Gtk.Enums.Gtk_Orientation;
--Pour modifier l'orientation du séparateur
procedure Set_Orientation(Self : access Gtk_Separator_Record;
Orientation : Gtk.Enums.Gtk_Orientation);
```

Les deux orientations possibles étant Orientation_Horizontal et Orientation_Vertical.

Les flèches

Fiche d'identité

- **Widget**: Gtk_Arrow.
- **Package** : Gtk.Arrow
- **Descendance** : GTK_Widget >> GTK_Misc
- **Description** : Ce widget est une simple flèche triangulaire.



Méthodes

Il y a fort peu de méthode pour ce type de widget. Voici le constructeur et le modifieur :

Code : Ada

```
procedure Gtk_New(Arrow      : out Gtk_Arrow;
                  Arrow_Type   : Gtk.Enums.Gtk_Arrow_Type;
                  Shadow_Type : Gtk.Enums.Gtk_Shadow_Type);
procedure Set(Arrow : access Gtk_Arrow_Record;
             Arrow_Type : Gtk.Enums.Gtk_Arrow_Type;
             Shadow_Type : Gtk.Enums.Gtk_Shadow_Type);
```

Le paramètre Arrow_Type indique l'orientation de la flèche (haut, bas, gauche, droite). Il peut prendre l'une des valeurs suivantes : Arrow_Up, Arrow_Down, Arrow_Left, Arrow_Right. Quant au paramètre Shadow_Type il indique le type d'ombrage et peut prendre les valeurs suivantes : Shadow_None, Shadow_In, Shadow_Out, Shadow_Etched_In, Shadow_Etched_Out.

Le calendrier

Fiche d'identité

- **Widget**: Gtk_Calendar.
- **Package** : Gtk.Calendar
- **Descendance** : GTK_Widget
- **Description** : Ce widget affiche une grille des jours du mois et de l'année en cours.



Méthodes

Le constructeur est simplissime donc je ne l'évoque pas. Toutefois, vous pouvez personnaliser votre calendrier :

Code : Ada

```
function Get_Display_Options  (Calendar : GtK_Calendar_Record) return GtK_Calendar_Display_Options; access
procedure Set_Display_Options  (Calendar : GtK_Calendar_Record; Flags : GtK_Calendar_Display_Options); access
```

Lorsque vous utilisez la méthode `Set_Display_Options()`, vous devez indiquer toutes les options désirées en les additionnant. Par exemple, `Set_Display_Options(option1 + option3)` configura le calendrier avec les options 1 et 3 mais pas avec la 2, même si celle-ci existait par défaut. Voici 5 options différentes :

- `Show_Heading`: affiche le mois et l'année ;
- `Show_Day_Names`: affiche les trois premières lettres du jour ;
- `No_Month_Change`: empêche l'utilisateur de changer de mois et donc d'année ;
- `Show_Week_Numbers`: affiche le numéro de la semaine sur la gauche du calendrier ;
- `Week_Start_Monday`: commence la semaine le lundi au lieu du dimanche.

D'autres méthodes sont disponibles :

Code : Ada

```
procedure Get_Date(Calendar : access GtK_Calendar_Record; Year : out Guint; Month : out Guint; Day : out Guint);
function Mark_Day (Calendar : access GtK_Calendar_Record; Day : Guint) return Boolean;
procedure Select_Day (Calendar : access GtK_Calendar_Record; Day : Guint);
function Select_Month (Calendar : access GtK_Calendar_Record; Month : Guint; Year : Guint) return Boolean;
```

La méthode `Get_Date()` vous permet de récupérer le jour, le mois et l'année sélectionnée par l'utilisateur. `Mark_Day()` vous permet de marquer en gris certains jours du mois, indiqués par leur numéro (0 signifiant aucun jour). Enfin, les méthodes `Select_Day()` et `Select_Month()` vous permettent comme leur nom l'indique de sélectionner un jour ou un mois et une année particulières.

Signaux

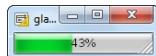
Les signaux disponibles sont les suivants :

- `Signal_Day_Selected / "day-selected"`: émis lorsque l'utilisateur clique sur un jour ;
- `Signal_Day_Selected_Double_Click / "day-selected-double-click"`: émis lorsque l'utilisateur double-clique sur un jour ;
- `Signal_Month_Changed / "month-changed"`: émis lorsque l'utilisateur change le mois ;
- `Signal_Next_Month / "next-month"`: émis lorsque l'utilisateur augmente le mois ;
- `Signal_Next_Year / "next-year"`: émis lorsque l'utilisateur augmente l'année ;
- `Signal_Prev_Month / "prev-month"`: émis lorsque l'utilisateur diminue le mois ;
- `Signal_Prev_Year / "prev-year"`: émis lorsque l'utilisateur diminue l'année.

Les barres de progression

Fiche d'identité

- Widget:Gtk_Progress_Bar.
- Package : GtK_Progress_Bar
- Descendance : GTK_Widget >> GTK_Progress
- Description : Ce widget est une barre affichant la progression d'un téléchargement, d'une installation...



La barre de progression

Méthodes

Le constructeur des GTK_Progress_Bar est sans intérêt. Passons directement aux autres méthodes que vous devriez déjà connaître puisque nous les avons déjà vu avec les GTK_Entry :

Code : Ada

```
procedure Set_Text(Progress_Bar : access GtK_Progress_Bar_Record; Text : UTF8_String);
function Get_Text(Progress_Bar : access GtK_Progress_Bar_Record) return UTF8_String;
-----  
procedure Set_Fraction(Progress_Bar : access GtK_Progress_Bar_Record; Fraction : Gdouble);
function Get_Fraction(Progress_Bar : access GtK_Progress_Bar_Record) return Gdouble;
-----  
procedure Set_Pulse_Step(Progress_Bar : access GtK_Progress_Bar_Record; Step : Gdouble);
function Get_Pulse_Step(Progress_Bar : access GtK_Progress_Bar_Record) return Gdouble;
-----  
procedure Pulse (Progress_Bar : access GtK_Progress_Bar_Record);
-----  
procedure Set_Orientation(Progress_Bar : access GtK_Progress_Bar_Record; Orientation : GtK_Progress_Bar_Orientation);
function Get_Orientation(Progress_Bar : access GtK_Progress_Bar_Record) return GtK_Progress_Bar_Orientation;
```

La méthode `set_text()` permet de définir le texte affiché sur la barre de progression (en général, il s'agit du pourcentage de progression). Les GTK_Progress_Bar ne fonctionnent pas à l'aide d'un minimum et d'un maximum, mais à l'aide d'un pourcentage de progression appelé `Fraction` et de type `Gdouble`. Pour rappel, `Set_Fraction()` fixe ce pourcentage de progression tandis que `Pulse()` augmente de la valeur du pas (`Step`). Ce pas peut être défini avec `Set_Pulse_Step()`.

Enfin, il est possible d'orienter la barre de progression : horizontale ou verticale ; progressant de gauche à droite ou de droite à gauche, de bas en haut ou de haut en bas. Les valeurs possibles des `GtK_Progress_Bar_Orientation` sont : `Progress_left_To_Right`, `Progress_Right_To_Left`, `Progress_Bottom_To_Top`, `Progress_Top_To_Bottom`.

En résumé :

- Pour afficher du texte, utilisez les `GTK_Label`.
- Pour afficher une image, utilisez les `GTK_Image`.
- Pour saisir du texte, utilisez les `GTK_GEntry` ou les `GTK_Text_View`.
- Pour saisir un nombre, utilisez les `GTK_Spin_Button` ou les `GTK_Scale`.
- N'hésitez pas à fouiner dans les divers packages de GTKAda pour découvrir votre bonheur ou allez sur le site d'Adacore pour consulter le [manuel de référence](#).

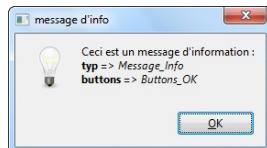
Les Widgets II : les boîtes de dialogue

Nous continuons le chapitre précédent. Cette fois, nous nous attaquons à des widgets plus complexes, à savoir les boîtes de dialogue. Vous savez, toutes ces petites fenêtres qu'un logiciel peut ouvrir pour vous rappeler de sauvegarder avant de quitter, pour vous demander quel fichier ouvrir ou comment vous allez paramétrer le logiciel. Toutes ces mini-fenêtres, simples ou complexes, s'appellent des **boîtes de dialogue**. Attention, leur fonctionnement est un peu plus complexe que celui des simples widgets vus jusque là donc je vous conseille d'être attentif.

Message

Fiche d'identité

- **Widget**: `Gtk_Message_Dialog`.
- **Package** : `GtkMessage_Dialog`
- **Désendance** : `GTK_Widget >> GTK_Container >> GTK_Bin >> GTK_Window >> GTK_Dialog`
- **Description** : Cette boîte de dialogue permet l'affichage de messages courts : erreur, avertissement, information...



Méthodes

Contrairement aux widgets précédents, la méthode la plus complexe est ici le constructeur dont voici la spécification :

Code : Ada

```
procedure Gtk_New(Dialog : out Gtk_Message_Dialog;
  Parent : Gtk.Window.Gtk.Window := null;
  Flags : Gtk.Dialog.Gtk.Dialog.Flags := 0;
  Typ : Gtk.Message_Type := Message_Info;
  Buttons : Gtk.Buttons_Type := Buttons_Close;
  Message : String);
```

Une procédure `Gtk_New_With_Markup()` identique existe si jamais vous souhaitez utiliser des balises pour la mise en forme. Le paramètre `Dialog` est bien entendu votre boîte de dialogue. Le paramètre `Parent` correspond à la fenêtre principale, la fenêtre mère. Le paramètre `Message` correspond au message indiqué dans la boîte de dialogue. Venons-en maintenant aux trois autres paramètres. `Flags` peut prendre les valeurs suivantes :

- `Modal` : si vous souhaitez désactiver la fenêtre mère le temps que votre boîte de dialogue est active. (la plus utile)
- `Destroy_With_Parent` : pour indiquer que la fermeture de la fenêtre mère détruira la boîte de dialogue. Mais par défaut, ce paramètre devrait être actif quoi que vous fassiez.
- `0` : si vous ne voulez indiquer aucun paramètre.



Ces valeurs sont situées dans le package `Gtk.Dialog`.

Le paramètre `Typ` indique le type de boîte de dialogue et donc l'image qui sera affichée (les valeurs possibles sont `Message_Info`, `Message_Warning`, `Message_Question`, `Message_Error`) ; les paramètres `Buttons` indiquent quant à lui le type et le nombre de boutons de la boîte (`Buttons_None`, `Buttons_Ok`, `Buttons_Close`, `Buttons_Cancel`, `Buttons_Yes_No`, `Buttons_Ok_Cancel`). Pour y voir plus clair, voici quelques exemples :



N'oubliez pas d'afficher votre boîte de dialogue grâce à `show_all` ! Enfin, pour personnaliser davantage votre boîte de dialogue, vous pouvez modifier le titre de la fenêtre grâce à la méthode `set_title()` disponible pour les `GTK_Window`, modifier l'image grâce à `set_image()` ou ajouter un texte secondaire grâce à `Format_Secondary_Text()` et `Format_Secondary_Markup()` :

Code : Ada

```
procedure Format_Secondary_Markup(Message_Dialog : access Gtk_Message_Dialog_Record;
  Message : String);
procedure Format_Secondary_Text (Message_Dialog : access Gtk_Message_Dialog_Record;
  Message : String);
function Get_Image(Dialog : access Gtk_Message_Dialog_Record)
return Gtk.Widget.Gtk.Widget;
procedure Set_Image(Dialog : access Gtk_Message_Dialog_Record;
  Image : access Gtk.Widget.Gtk.Widget_Class);
```

Les signaux

Pour utiliser une boîte de dialogue, nous n'allons pas utiliser de signaux. Nous allons utiliser la fonction `run()` présente dans le

```
package GTK.Dialog;
Code : Ada
function Run(Dialog : access Gtk_Dialog_Record) return
    Gtk_Response_Type;
```

Cette fonction affiche la boîte de dialogue (comme `show` ou `show_all`), crée une sorte de «mini-boucle infinie» et renvoie la réponse de l'utilisateur (a-t-il cliqué sur OK ou Annuler?). Voici un exemple d'utilisation :

```
Code : Ada
...
Win : GTK_Window ;
Msg : GTK_Message_Dialog ;
BEGIN
    --Création de la boîte de dialogue Msg
    GtK_New(Msg, Win, 0, Message_Error, Buttons_Ok_Cancel, "Ceci est
    un message d'erreur") ;
    --Affichage de la fenêtre principale
    Win.Show All ;
    --Affichage et attente d'une réponse de la boîte de dialogue
    IF Msg.Run = GtK_Response_Ok
    THEN
        --écrire ici les instructions
    END IF ;
...
```

Le code ci-dessus crée une boîte de dialogue de type Erreur. L'utilisateur a deux boutons accessibles : OK et Cancel. Si il clique sur OK, `run()` renvoie comme `Gtk_Response_Type` la valeur suivante : `Gtk_Response_OK`. Le programme n'a alors plus qu'à gérer cette réponse. Voici une liste de réponses possibles :

- `Gtk_Response_Delete_Event`: si l'utilisateur clique sur l'icône de fermeture de la barre de titre ;
- `Gtk_Response_Ok`: si l'utilisateur appuie sur le bouton `Buttons_Ok` ;
- `Gtk_Response_Cancel`: si l'utilisateur appuie sur le bouton `Buttons_Cancel` ;
- `Gtk_Response_Close`: si l'utilisateur appuie sur le bouton `Buttons_Close` ;
- `Gtk_Response_Yes`: si l'utilisateur appuie sur le bouton `Yes` de `Buttons_Yes_No` ;
- `Gtk_Response_No`: si l'utilisateur appuie sur le bouton `No` de `Buttons_Yes_No` ;

 Mais... ma boîte de dialogue ne se ferme pas toute seule ! 

Vous devrez donc, lorsque vous traitez les réponses possibles, penser à ajouter une instruction pour détruire la boîte de dialogue. Cette instruction est présente dans le package `GTK.Widget` (et est donc disponible pour n'importe quel widget, même les boutons, images ou conteneurs) :

```
Code : Ada
procedure Destroy(Widget : access Gtk_Widget_Record);
```

À propos

Fiche d'identité

- **Widget**: `Gtk_About_Dialog`.
- **Package**: `GtkAbout_Dialog`
- **Descendance**: `GTK_Widget` >> `GTK.Container`
- **Description** : Ce widget est souvent utilisé lorsque l'utilisateur clique sur le menu «*À propos*» afin d'afficher le nom du logiciel, sa version, son auteur, son logo...



Méthodes

Le constructeur des `GTK_About_Dialog` est simplissime et ne prend qu'un seul paramètre. Concentrons nous donc sur l'apparence de cette boîte de dialogue :

```
Code : Ada
function Get_Comments (About : access Gtk_About_Dialog_Record)
return UTF8_String;
procedure Set_Comments(About : access Gtk_About_Dialog_Record;
Comments : UTF8_String);

function Get_Copyright (About : access Gtk_About_Dialog_Record)
return UTF8_String;
procedure Set_Copyright(About : access Gtk_About_Dialog_Record;
Copyright : UTF8_String);

function Get_License (About : access Gtk_About_Dialog_Record)
return UTF8_String;
procedure Set_License(About : access Gtk_About_Dialog_Record;
License : UTF8_String);

function Get_Program_Name (About : access Gtk_About_Dialog_Record)
return UTF8_String;
procedure Set_Program_Name(About : access Gtk_About_Dialog_Record;
Name : UTF8_String);

function Get_Version (About : access Gtk_About_Dialog_Record)
return UTF8_String;
procedure Set_Version(About : access Gtk_About_Dialog_Record;
Version : UTF8_String);

function Get_Website (About : access Gtk_About_Dialog_Record) return
    UTF8_String;
procedure Set_Website(About : access Gtk_About_Dialog_Record;
Website : UTF8_String);

function Get_Website_Label (About : access Gtk_About_Dialog_Record)
return UTF8_String;
procedure Set_Website_Label(About : access
    Gtk_About_Dialog_Record;
Website_Label : UTF8_String);
```

Les méthodes ci-dessus permettent d'obtenir ou de définir les commentaires (description succincte du logiciel), le copyright, la licence, le nom du programme, sa version actuelle, l'adresse du site web correspondant ainsi que le nom du site web (qui sera affiché à la place de l'adresse). Toutes ces méthodes manipulent des `UTF8_String` et il est possible de les mettre en forme comme nous l'avons vu avec les étiquettes. A noter que la création d'une licence engendre automatiquement la création du bouton du même nom. Si vous cliquez sur celui-ci, une seconde fenêtre de dialogue s'ouvrira. Il est également possible d'ajouter un troisième bouton, en plus des boutons `License` et `Close` : le bouton `Credits`.

```
Code : Ada
```

```

function Get_Artists (About : access Gtk_About_Dialog_Record)
return GNAT.Strings.String_List;
procedure Set_Artists(About : access Gtk_About_Dialog_Record;
                      Artists : GNAT.Strings.String_List);

function Get_Authors (About : access Gtk_About_Dialog_Record)
return GNAT.Strings.String_List;
procedure Set_Authors(About : access Gtk_About_Dialog_Record;
                      Authors : GNAT.Strings.String_List);

function Get_Documenters (About : access Gtk_About_Dialog_Record)
return GNAT.Strings.String_List;
procedure Set_Documenters(About : access Gtk_About_Dialog_Record;
                          Documenters : GNAT.Strings.String_List);

function Get_Translator_Credits (About : access Gtk_About_Dialog_Record)
return UTF8_String;
procedure Set_Translator_Credits(About : access Gtk_About_Dialog_Record;
                                 Translator_Credits : UTF8_String);

```

Les méthodes ci-dessus permettront de définir les artistes, les auteurs, les personnes ayant documenté le logiciel ainsi que les éventuels traducteurs. Les trois premières méthodes utilisent des `GNAT.Strings.String_List` qui ne sont rien de plus que des tableaux de pointeurs vers des strings. Voici par exemple comment renseigner la liste des artistes :

Code : Ada

```

...
List : String_List(1..2) ;
BEGIN
...
List(1) := NEW_String("Picasso") ;
List(2) := NEW_String("Rembrandt") ;
Ma_Boite_De_Dialogue.Set_Artists(List) ;
...

```

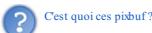
Si vous souhaitez consulter les spécifications du type `String_List`, ouvrez le package `System.String` (`s-string.ads`) car `GNAT.String` n'est qu'un surnommage. Enfin, vous pouvez également définir un logo à votre boîte de dialogue (en général, le logo du logiciel) :

Code : Ada

```

function Get_Logo (About : access Gtk_About_Dialog_Record) return
Gdk_Pixbuf.Gdk_Pixbuf;
procedure Set_Logo(About : access Gtk_About_Dialog_Record;
                  Logo : access
Gdk_Pixbuf.Gdk_Pixbuf_Record'Class);

```



C'est quoi ce pixbuf ?

Les `Gdk_Pixbuf_Record` sont en fait des images gérées de façon bas niveau par Gdk. Comme je ne souhaite pas perdre de temps supplémentaire sur les images, nous nous contenterons donc d'utiliser une des méthodes `Get()` prévues pour les `GTK_Image`. Celle-ci extrait un `Gdk_Pixbuf` d'une `Gtk_Image` :

Code : Ada

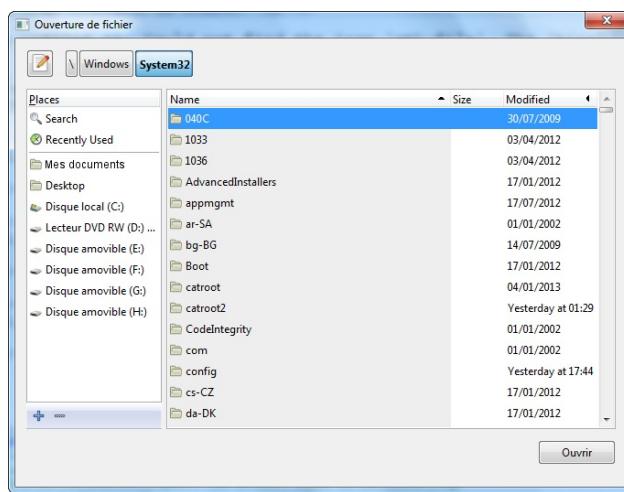
```

...
Image : GTK_Image ;
BEGIN
...
Gtk_New(Image,"adresse/du/fichier/image") ;
Ma_Boite_De_Dialogue.set_logo(img.get) ;
...
```

Sélection de fichier

Fiche d'identité

- **Widget:Gtk_File_Chooser_Dialog et Gtk_File_Chooser_Dialog.**
- **Package:** `Gtk_File_Chooser_Dialog` à `Gtk.File_Chooser_Dialog`
- **Descendance :** `GTK_Widget`>>`GTK.Container`>>`GTK_Bin`>>`GTK_Window`>>`GTK_Dialog`
- **Description :** Ce widget permet de sélectionner un fichier ou un répertoire pour l'ouvrir ou l'enregistrer.



Boîte de dialogue : Sélection de fichier

Méthodes des `GTK_File_Chooser_Dialog`

Les `GTK_File_Chooser_Dialog` ne sont pas compliqués à créer :

Code : Ada

```

procedure Gtk_New (Dialog : out Gtk_File_Chooser_Dialog;
                  Title : String;
                  Parent : access
Gtk.Window.Gtk_Window_Record'Class;
                  Action : Gtk.File_Chooser.File_Chooser_Action);

```

Le paramètre `Title` est bien entendu le titre de votre boîte de dialogue ("ouverture du fichier de sauvegarde" par exemple). `Parent` est le nom de la fenêtre parente, la fenêtre principale. Quant au paramètre `Action`, il indique le type de fenêtre désiré. Il peut prendre l'une des valeurs suivantes :

- Action_Open : fenêtre pour ouvrir un fichier.
- Action_Save : fenêtre pour sauvegarder un fichier. Elle comporte une ligne pour indiquer le nom du fichier à créer ainsi qu'un bouton pour créer un sous-répertoire.
- Action_Select_Folder : fenêtre pour ouvrir un répertoire. Les fichiers sont automatiquement grisés.
- Action_Create_Folder : fenêtre pour créer un répertoire.

Et c'est à peu près tout ce que vous pouvez faire avec une `GTK_File_Chooser_Dialog`.

Les `GTK_File_Chooser` (sans le Dialog)



Mais... il n'y a aucun bouton Ouvrir, Annuler ou Enregistrer ! À quoi ça sert ?

Les méthodes des `GTK_File_Chooser_Dialog` sont quelque peu limitées. Pour la personnaliser un peu (notamment en ajoutant quelques boutons), vous devrez récupérer le `GTK_File_Chooser` de votre `GTK_File_Chooser_Dialog`, car c'est lui que vous pourrez personnaliser. Pour effectuer la récupération, une méthode `"+"` est disponible dans le package `GTK.File_Chooser_Dialog.Exemple`:

Code : Ada

```
...
  Chooser : GTK_File_Chooser ;
  Dialog : GTK_File_Chooser_Dialog ;
BEGIN
...
  Chooser := + Dialog ; --Chooser pointera vers le GTK_File_Chooser
  contenu dans Dialog
...
```

Passons maintenant aux méthodes liées aux `GTK_File_Chooser`. Pour commencer, nous voudrions ajouter au moins un bouton à notre boîte de dialogue.

Code : Ada

```
procedure Set_Extra_Widget (Chooser : Gtk_File_Chooser;
                           Extra_Widget : Gtk_Widget_Record'Class);
                           access
```

La méthode `Set_Extra_Widget()` permet d'ajouter n'importe quel type de widget au bas de votre fenêtre : un bouton ou mieux ! Une boîte à boutons ! Maintenant voyons comment paramétrer notre boîte de dialogue :

Code : Ada

```
procedure Set_Action(Chooser : Gtk_File_Chooser;
                     Action : File_Chooser_Action);
function Get_Action (Chooser : Gtk_File_Chooser) return
File_Chooser_Action;
-----
procedure Set_Select_Multiple(Chooser : Gtk_File_Chooser;
                             Select_Multiple : Boolean);
function Get_Select_Multiple (Chooser : Gtk_File_Chooser) return
Boolean;
-----
procedure Set_Show_Hidden(Chooser : Gtk_File_Chooser;
                         Show_Hidden : Boolean);
function Get_Show_Hidden (Chooser : Gtk_File_Chooser) return
Boolean;
-----
procedure Set_Current_Name(Chooser : Gtk_File_Chooser;
                           Name : UTF8_String);
function Set_Current_Folder(Chooser : Gtk_File_Chooser;
                           Filenamne : String) return Boolean;
function Get_Current_Folder(Chooser : Gtk_File_Chooser) return
String;
function Get_Filename (Chooser : Gtk_File_Chooser) return String;
```

La méthode `Set_Action()` vous permet de redéfinir le type de fenêtre (sauvegarde, ouverture de fichier ou de répertoire) si jamais vous ne l'avez pas déjà fait. La méthode `Set_Select_Multiple()` permet de définir si l'utilisateur peut sélectionner un unique fichier (le paramètre `select_multiple` valant `TRUE`) ou bien plusieurs (`select_multiple` valant `FALSE`). De la même manière, `Set_Show_Hidden()` permet d'afficher les fichiers cachés (si `Show_Hidden` = `TRUE`) ou pas. La méthode `Set_Current_Name()` n'est utile que pour des boîtes de dialogue de sauvegarde car elle permet de définir un nom par défaut à votre sauvegarde. Cette méthode peut vous être très utile pour des sauvegardes de partie en fournissant un titre par défaut, par exemple du genre "Nomdujour-jour-mois-annee.sav". De la même manière, `Set_Current_Folder()` permet de définir un répertoire par défaut à l'ouverture de la boîte de dialogue : très utile notamment si vous souhaitez que l'utilisateur enregistre ses sauvegardes dans un même répertoire prévu à cet effet. Enfin, la méthode `Get_Filename()` permettra à vos callbacks d'obtenir l'adresse du fichier sélectionné afin de pouvoir l'ouvrir. Attention, il s'agit de l'adresse absolue et non relative du fichier, c'est à dire du chemin complet.



Et si je veux ouvrir plusieurs fichiers en même temps ?

Là, les choses se compliquent un petit peu. Vous utiliserez la méthode suivante :

Code : Ada

```
function Get_Filenames(Chooser : Gtk_File_Chooser) return
Gtk.Enums.String_SList.GSlist;
```

Vous aurez sûrement remarqué le `S` supplémentaire dans le nom de la méthode ainsi que le type du résultat : `Gtk.Enums.String_SList.GSlist`. Cela signifie que vous devrez utiliser le package `Gtk.Enums` et ajouter une clause « `USE Gtk.Enums.String_SList` » (car il s'agit d'un sous-package interne à `Gtk.Enums`) pour utiliser ces `GSlist`. Qu'est-ce qu'une `GSlist` ? C'est la contraction de « *generic single-linked list* », soit « *liste générique simplement chaînée* ». Vous devriez comprendre ce que cela signifie désormais.

Vous trouverez les méthodes associées dans le package `Glib.GSlist`. Notamment, vous aurez besoin des méthodes ci-dessous. La première, `Length()` renvoie la longueur de la chaîne (celles-ci étant indexées de 0 à «Longueur-1») tandis que la seconde renvoie la donnée numéro N de la `GSlist`.

Code : Ada

```
function Length (List : in GSlist)
                return Uint;
function Nth_Data (List : in GSlist;
                  N : in Uint)
                return Gpointer;
```

Voici un exemple d'utilisation de ces méthodes avec un callback appelé lorsque l'utilisateur appuie sur le bouton OK :

Code : Ada

```
PROCEDURE OK_Action(Emetteur : ACCESS GTK_Button_Record'Class;
                     F : Gtk_File_Chooser_Dialog) IS
  PRAGMA Unreferenced(Emetteur);
  USE Gtk.Enums.String_SList;
  L : GSlist;
BEGIN
  L := Get_Filenames(+F); --On récupère les noms de
  fichiers du file_chooser de F
  FOR I IN 0..Length(L)-1 LOOP --On affiche les noms de la
  liste.
    Put_Line(Nth_Data(L,I)); --Attention, elle est indexée à
    partir de 0 ! Il y a donc
  END LOOP;
  F.destroy; --Un décalage dans les indices
  --Enfin on détruit la boîte de
dialogue
END Ok_Action;
```

Deux widgets supplémentaires

Enfin, il est possible d'ajouter deux derniers widgets : un prévisualiseur de fichier et un filtre. Le prévisualiseur est un widget de votre cru affichant une miniature du contenu du fichier sélectionné. Par exemple, certains logiciels de graphisme affichent une miniature de vos photos avant que vous ne les ouvriez pour de bon. Cela peut facilement s'ajouter à l'aide des méthodes :

Code : Ada

```
procedure Set_Preview_Widget(Chooser : Gtk_File_Chooser;
                             Preview_Widget : access
                               Gtk.Widget.Gtk_Widget_Record'Class);
function Get_Preview_Widget(Chooser : Gtk_File_Chooser) return
                               Gtk.Widget.Gtk_Widget;
```

Les filtres, quant à eux permettent de filtrer les fichiers, de n'en afficher qu'une partie seulement selon des critères :



Pour cela vous devrez créer un filtre, c'est-à-dire un widget de type `Gtk_File_Filter` disponible via le package `Gtk.File_Filter`. Vous y trouverez les méthodes ci-dessous :

Code : Ada

```
--le constructeur, sans commentaires
procedure Gtk_New(Filter : out Gtk_File_Filter);
--pour définir le texte à afficher
procedure Set_Name(Filter : access Gtk_File_Filter_Record; Name :
String);
--pour définir les critères, nous allons voir celui-ci plus en
détail
procedure Add_Pattern(Filter : access Gtk_File_Filter_Record;
                      Pattern : String);
```

Je ne m'attarde pas sur les deux premières méthodes, plutôt évidentes. En revanche, la troisième est importante : c'est elle qui fixe la règle d'affichage. Ces règles correspondent aux règles utilisées sous la console. Exemple :

Code : Ada

```
Filtre1.Add_Pattern("*.") ; --affiche tous les fichiers (*
signifie n'importe quel caractère)
Filtre2.Add_Pattern("*.txt") ; --n'affiche que les fichiers se
finissant par ".txt"
```

Une fois vos filtres créés, vous pourrez les ajouter ou les supprimer de vos `Gtk_File_Chooser` grâce aux méthodes :

Code : Ada

```
procedure Add_Filter  (Chooser : Gtk_File_Chooser;
                      Filter : access
                        Gtk.File_Filter.Gtk_File_Filter_Record'Class);
procedure Remove_Filter(Chooser : Gtk_File_Chooser;
                       Filter : access
                         Gtk.File_Filter.Gtk_File_Filter_Record'Class);
```

Signaux des `Gtk_File_Chooser`

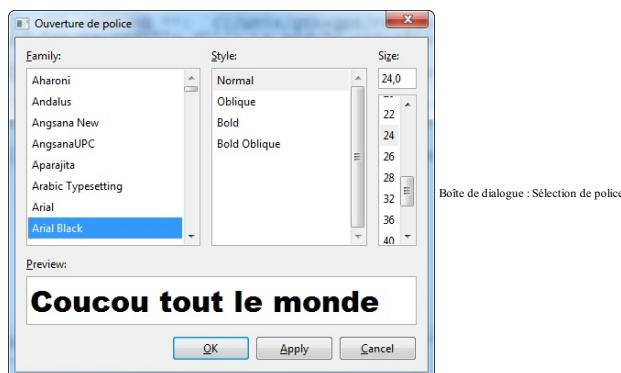
Les `Gtk_File_Chooser_Dialog` n'apportent pas de nouveaux signaux. En revanche, les `Gtk_File_Chooser` ont quelques signaux utiles :

- `Signal_Current_Folder_Changed / "current-folder-changed"` : émis lorsque l'utilisateur change de dossier;
- `Signal_File_Activated / "file-activated"` : émis lorsque l'utilisateur double-clique sur un fichier ou presse la touche entrée;
- `Signal_Selection_Changed / "selection-changed"` : émis lorsque l'utilisateur modifie sa sélection ;
- `Signal_Update_Preview / "update-preview"` : émis lorsque le prévisualiseur doit être mis à jour.

Sélection de police

Fiche d'identité

- **Widget:** `Gtk_Font_Selection_Dialog` et `Gtk_Font_Selection`.
- **Package:** `GtkFont_Sélection_Dialog` ou plus directement `GtkFont_Selection`.
- **Descendance:** `GTK_Widget` >> `GTK_Container` >> `GTK_Bin` >> `GTK_Window` >> `GTK_Dialog`.
- **Description:** Cette boîte de dialogue permet de sélectionner une police, une taille et un style. Elle affiche également un aperçu de la mise en forme obtenue.



Boîte de dialogue : Sélection de police

Méthodes

Les `Gtk_Font_Selection_Dialog` présentent assez peu de méthodes. Elles peuvent se résumer à ceci :

Code : Ada

```
procedure Gtk_New(Widget : out Gtk_Font_Selection_Dialog;
                  Title : UTF8_String);
function Set_Font_Name(Fsd : access
                           Gtk_Font_Selection_Dialog_Record;
                           Fontname : String) return Boolean;
function Get_Font_Name(Fsd : access
                           Gtk_Font_Selection_Dialog_Record) return String;
```

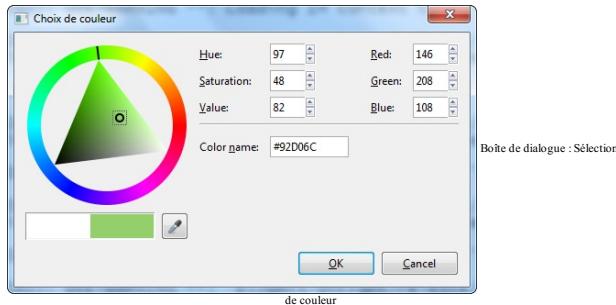
Le constructeur n'exige rien de plus qu'un titre pour votre fenêtre. Ensuite, la méthode `Get_Font_Name()` vous permet de récupérer une chaîne de caractères contenant le nom de la police suivi éventuellement du style (gras, italique...) puis de la taille.

i A noter que les `Gtk_Font_Selection_Dialog` et `Gtk_Color_Selection` définis dans le même package ont des méthodes identiques. La différence réside dans le fait qu'ils peuvent être insérés dans des fenêtres plus complexes.

Sélection de couleur

Fiche d'identité

- `Widget:Gtk_Color_Selection_Dialog` et `Gtk_Color_Selection`.
- `Package : GtkColor_Selection_Dialog` et `GtkColor_Selection`.
- `Descendance : GTK_Widget >> GTK_Container >> GTK_Bin >> GTK_Window >> GTK_Dialog`.
- `Description : Cette boîte de dialogue permet de sélectionner une couleur.`



Boîte de dialogue : Sélection

de couleur

Méthodes

Là encore, les méthodes sont très limitées. Le constructeur ne demande qu'un titre pour votre boîte de dialogue. En revanche, pour manipuler ce widget, vous devrez récupérer son `Gtk_Color_Selection` à l'aide de la méthode `Get_Colordsel()` :

Code : Ada

```
function Get_Colordsel(Color_Selection_Dialog : access
                        Gtk_Color_Selection_Record)
  return Gtk.Color_Selection.Gtk_Color_Selection;
```

Une fois le `Gtk_Color_Selection` récupéré, vous pourrez connaître ou fixer la couleur actuelle à l'aide des deux méthodes suivantes :

Code : Ada

```
--Fixer ou obtenir la couleur actuelle
procedure Set_Current_Color(Colordsel : access
                            Gtk_Color_Selection_Record);
procedure Get_Current_Color(Colordsel : access
                            Gtk_Color_Selection_Record);
--Fixer ou obtenir l'ancienne couleur
procedure Set_Previous_Color(Colordsel : access
                            Gtk_Color_Selection_Record);
procedure Get_Previous_Color(Colordsel : access
                            Gtk_Color_Selection_Record);
Color      : out Gdk.Color.Gdk_Color);
```

La couleur obtenue se présente sous la forme d'une variable composite `Gdk_Color` (oui je sais, c'est notre première variable `Gdk` 😊). Pour la manipuler, utilisez le package `Gdk.Color` et les méthodes suivantes :

Code : Ada

```
procedure Set_Rgb (Color : out Gdk_Color; Red, Green, Blue :
                    Uint16);
function Red (Color : Gdk_Color) return Uint16;
function Green (Color : Gdk_Color) return Uint16;
function Blue (Color : Gdk_Color) return Uint16;
function To_String (Color : Gdk_Color) return String;
```

`Set_Rgb()` vous permettra de définir votre propre couleur à partir de ses valeurs RGB (Rouge Vert et Bleu). Attention, celles-ci sont définies de 0 à 65535 et pas de 0 à 255. À l'inverse, les fonctions `Red()`, `Green()`, `Blue()` vous permettront de connaître chacune des composantes de votre couleur, tandis que `To_String()` vous renverra le code hexadécimal HTML de votre couleur (#92D06C dans l'exemple ci-dessus).

Signaux

Le seul signal nouveau apporté par les `Gtk_Color_Selection` est le `signalSignal_Color_Changed / "color_changed"` indiquant que l'utilisateur a changé la couleur.

Cas général

Fiche d'identité

- `Widget:Gtk_Dialog`.
- `Package : Gtk`.
- `Descendance : GTK_Widget >> GTK_Container >> GTK_Bin >> GTK_Window`.
- `Description : Il s'agit là de la boîte de dialogue mère, personnalisable à volonté. Par défaut, Gtk considère qu'une boîte de dialogue est une fenêtre coupée en deux. La partie haute comprend divers widgets, tandis que la partie basse, appelée Zone d'actions, comprend des boutons rangés horizontalement.`

Méthodes

Nous avons déjà vu bon nombre de méthodes des `GTK_Dialog` comme les constructeurs ou la méthode `run()`, mais en voici quelques-unes qui pourraient vous être utile pour personnaliser vos boîtes de dialogue (y compris celles vues précédemment) :

Code : Ada

```
function Get_Action_Area (Dialog : access Gtk_Dialog_Record) return
  Gtk.Box.Gtk_Box;
function Get_Content_Area(Dialog : access Gtk_Dialog_Record) return
  Gtk.Box.Gtk_Box;
function Get_Vbox (Dialog : access Gtk_Dialog_Record) return
  Gtk.Box.Gtk_Box;
function Add_Button(Dialog : access Gtk_Dialog_Record;
                   Text : UTF8_String;
                   Response_Id : Gtk_Response_Type) return
  Gtk.Widget.Gtk_Widget;
```

Les trois premières méthodes permettent respectivement de récupérer la zone d'action, la zone de contenu et le conteneur principal de votre boîte de dialogue afin de les personnaliser. La fonction `Add_Button()` crée un bouton, tout en l'ajoutant à la zone d'action de votre boîte de dialogue (pas besoin de `set_extra_widget()` par exemple). Le paramètre `Text` est bien entendu le texte du bouton. `Response_Id` est quant à lui le type de réponse qu'il renverra (ce sont les fameux `Gtk_Response` vus au début de cette partie).

Signaux

Enfin, voici deux signaux généraux aux boîtes de dialogue :

- `Signal_Close / "close"` : émis lorsque l'utilisateur appuie sur la touche Escap.
- `Signal_Response / "response"` : émis lorsque l'utilisateur clique sur l'un des widgets de la zone d'action (en général sur un bouton).

En résumé :

- Pour afficher messages, options ou informations sur votre logiciel, ne créez pas de nouvelles `GTK_Window`, mais utilisez plutôt les diverses boîtes de dialogue mises à votre disposition.
- Pour connaître l'action choisie par l'utilisateur, le résultat de la boîte de dialogue, n'utilisez pas les signaux mais la fonction `Run()`.
- La fonction `Run()` exécute automatiquement la boîte de dialogue alors évitez d'utiliser plusieurs `IF` ou `ELSIF` imbriqués. Privilégiez les `CASE`.

[TP] Le démineur

Nous voici rendu à notre premier TP en programmation évènementielle. Pour cette première, nous allons réaliser un jeu tout simple : un démineur ! Vous savez, ce bon vieux jeu livré avec la plupart des systèmes d'exploitation où vous devez placer des drapeaux sur des cases pour signaler la présence d'une mine et détruire les autres cases. Mais attention, si ce jeu est simple, il vous demandera toutefois un peu de temps et de concentration, ne le prenez pas à la légère.

Pour ceux qui trouve ce challenge trop ardu, pas d'inquiétude, je vais vous indiquer une démarche à suivre, comme toujours. De plus, nous ne terminerons pas notre programme : nous n'avons pas encore vu comment créer des menus par exemple. L'achèvement de ce programme constituera notre second TP de programmation évènementielle. Prêt ?

Règles du jeu et cahier des charges

Quelques rappels

Pour ceux qui auraient oublié ou qui n'auraient pas connu cette antiquité (toujours vivante) de l'informatique que constitue le jeu du démineur voici un petite capture d'écran :



Le jeu de démineur

Le jeu est simple : au début, vous disposez d'une grille de 9×9 cases (pour le niveau facile). En faisant un clic gauche sur l'une d'entre elles, elle s'autodétruit. Mais attention ! Si l'y a furet des 10 mines en dessous, elle explose et vous perdez la partie. S'il n'y en a pas, un chiffre apparaît à la place de la case indiquant le nombre de mines présentes juste à côté de cette case (au-dessus, au-dessous, à gauche ou à droite, mais aussi en diagonale soit un maximum de 8 bombes).

Si vous êtes certains d'avoir deviné l'emplacement d'une bombe, il suffit de faire un clic droit sur la case pour y placer un drapeau qui signalera sa présence et évitera que vous ne cliquez dessus. Vous gagnez lorsque toutes les cases non minées ont été détruites.

Je n'y ai jamais joué mais ça a l'air complexe. Comment peut-on gagner ?

C'est pourtant simple, prenons un exemple :

0	0	0	0
0	1	1	1
0	1	?	?
0	1	?	?

Le **1** à la deuxième ligne et deuxième colonne indique qu'il y a une seule mine dans son entourage, elle ne peut se situer qu'à la troisième ligne et troisième colonne. Par conséquent, les autres **1** nous indiquent qu'il ne peut pas y avoir d'autre mine parmi les points d'interrogation de la deuxième colonne ou de la deuxième ligne : nous pourrons cliquer dessus sans risque.

Règles retenues

Le démineur que je vous propose de réaliser conservera la plupart des règles :

- Il présentera un compteur de drapeaux restants.
- Les mines seront placées de façon aléatoire.
- Un clic gauche détruit une case.
- Un clic gauche sur une mine entraîne une défaite immédiate.
- Un clic gauche sur un drapeau est neutralisé (aucune action).
- Les cases ne jouant aucune mine n'afficheront rien (même pas de 0).
- Un clic gauche sur une case 0 entraînera automatiquement la destruction des 8 cases alentours, évitant à l'utilisateur de cliquer de nombreux fois pour découvrir des chiffres.
- Un clic droit sur une case normale placera ou enlève un drapeau.
- Un clic droit sur un drapeau ne placera pas de point d'interrogation.
- Les chiffres indiquant le nombre de mines auront chacun une couleur spécifique pour faciliter leur visualisation.
- Le démineur classique propose des fonctionnalités comme le clic gauche-droite qui permet de visualiser les cases concernées par un chiffre. Cette fonctionnalité n'est pas exigée.
- Le démineur réutilisera facilement par d'autres programmes.

Ce premier démineur sera encore rudimentaire. Nous entrerons les dimensions de la grille et le nombre de mines via la console. Dans le prochain TP, nous ajouterons des menus à notre démineur ainsi que des niveaux de difficultés intégrant directement les dimensions et le nombre de bombes. Pour l'heure, nous allons créer les packages nécessaires à la constitution d'une grille de démineur réutilisable facilement par d'autres programmes.

Quelques ressources

Pour réaliser ce TP, vous aurez besoin de quelques ressources. Les images tout d'abord. Faites un clic droit puis «enregistrer sous» :



J'ai utilisé la mine noire comme icône du jeu et la rouge lorsque le joueur cliquera sur une mine. Il est bien entendu évident que la mine rouge n'apparaît qu'en cas d'échec. Je vous joint également une police d'écriture rappelant les affichages à 7 bâtons de nos réveils pour votre compteur :

DS-DIGITAL

Police DS-Digital

Un petit coup de main Premier objectif : détruire une case

Votre premier problème sera l'action principale du démineur : détruire des boutons et les remplacer (ou pas) par des étiquettes ou des images. Je vous conseille donc de commencer par là. Créez une simple fenêtre contenant un unique bouton et un unique callback. Pour atteindre votre objectif, vous aurez besoin d'une méthode commune à tous les widgets : `destroy()`. Comme son nom l'indique, cela vous permettra de détruire votre bouton : votre `GTK_Button` sera réinitialisé à `NULL` et le bouton disparaîtra de votre fenêtre.

Pour ce qui est de l'affichage de texte ou d'image, je vous conseille de créer un type `TAGGED_RECORD` contenant la plupart des infos comme :

- La case est-elle minée ?
- Le nombre de bombes avoisinant cette case.
- L'état de la case : normal, crevée ou signalée par un drapeau.
- Les différents widgets correspondants : `GTK_Button`, `GTK_Label` et `GTK_Image`.

Vous serez sûrement amenés à élargir ces attributs au fur et à mesure mais cela devrait suffire pour l'instant. Revenons à l'affichage d'une image ou d'un texte. Celui-ci ne peut se faire qu'au moment de la destruction du bouton et donc au sein de votre callback. Ce dernier prendra un paramètre qui est votre case (que j'ai personnellement appelé `TYPE_T_Tile`, vu que le mot

CASE est un mot réservé : il sera donc au moins issu du package `Gtk.Handlers.User_Callback`. Je dis «au moins» car nous verrons qu'il faudra modifier cela avec le prochain problème. C'est également à ce moment que vous pourrez initialiser le `GTK_Label` et la `GTK_Image`, il est en effet inutile qu'ils encombrent la mémoire plus tôt.

Second objectif : placer un drapeau

Vous devrez ensuite être capable d'ajouter ou de supprimer une image dans un bouton. Cela se fait assez simplement avec les méthodes `Add()` et `Destroy()`. Commencez par réaliser cette action avec un simple clic gauche et n'oubliez pas d'afficher les images ou textes avec la méthode `Show()`.

Une fois cette étape franchie, vous devrez modifier votre callback précédent pour qu'il gère le clic gauche (destruction de case) et le clic droit (création/suppression d'un drapeau). Cela vous obligera à utiliser le package `Gdk.Event` comme nous l'avons vu dans le chapitre sur les signaux. Nous utiliserons notamment l'événement `Signal_Button_Press_Event` et la fonction `Get_Button()`. Pour rappel, cette fonction renvoie 1 si vous cliquez sur le bouton de gauche, 2 sur le bouton central et 3 sur le bouton de droite de votre souris.

Mais surtout, l'événement `Signal_Button_Press_Event` exigera que votre callback soit une fonction renvoyant un booléen et non une procédure. Vous devrez donc le modifier et faire appel au package `Gtk.Handlers.User_Return_Callback` plutôt qu'à `Gtk.Handlers.User_Callback`.

Troisième objectif : passer du bouton unique à la grille

Les choses restent simples tant que vous n'avez qu'un seul bouton à gérer. Mais elles vont se compliquer dès lors que vous devrez gérer de nombreux boutons. Je vous conseille de commencer par créer un type «tableau de cases». Cela n'est pas bien compliqué mais vous devez penser que votre callback devra à l'avenir être capable de modifier le compteur de drapeaux ou de découvrir les cases adjacentes. Il va très vite devenir nécessaire de créer un type spécifique pour englober toutes les informations et widget du jeu.

Ce nouveau type (appelons-le `T_Game` par exemple) sera le paramètre principal de votre callback. Je vous conseille de lui attribuer trois paramètres : la largeur, la longueur et le nombre de bombes. Cela vous permettra de l'initialiser à votre convenance et de le réutiliser facilement dans le prochain TP.

Quatrième objectif : destruction de cases en cascade

Pour l'heure vous ne devriez avoir la possibilité de détruire qu'une case à la fois. Or, si l'on clique sur une case nulle (pas de mine au-dessous ou aux alentours), on sait que les 8 cases alentours pourraient être détruites immédiatement.



Rien de bien compliqué ! Il suffit de deux boucles `FOR` imbriquées pour détruire ces cases. C'est de la rigolade.

Sauf que si l'une d'entre elles (ou plusieurs) sont également nulles, leurs entourages respectifs devraient également être détruits et ainsi de suite. Cela impliquera donc une sorte de destruction en chaîne des cases. Et cette destruction en chaîne peut se diriger dans n'importe quelle direction.



Ah oui... C'est plus compliqué que prévu. Mais peut-être qu'avec une troisième boucle...

Une troisième boucle pourrait être une idée seulement vous devrez très certainement faire de très nombreux tests inutiles et créer un algorithme compliqué. Non, le plus judicieux serait d'utiliser la récursivité. Votre double boucle `FOR` détruire les 8 cases entourant une case nulle et si une ou plusieurs d'entre elles sont également des cases nulles, le processus sera relancé pour ces cases (et celles-ci seulement).

Une solution possible

Les spécifications

Nous voici à l'heure fatigante de la solution. Un petit commentaire avant de commencer : pour garder une certaine uniformité avec GTKAda, vous constaterez que mon code-solution utilise régulièrement deux types :

- Un type `TAGGED RECORD` contenant les informations nécessaires. Celui-ci est toujours accompagné du suffixe `_Record`, par exemple `T_Game_Record`
- Un type `ACCESS` pointant sur mon type structuré. Celui-ci n'a pas de suffixe et est le type généralement utilisé par les fonctions et procédures.

Commençons à découvrir le code source de la solution par les spécifications. Tout d'abord voici le fichier définissant ce qu'est une case :

Code : Ada - P_Tile.ads

```
-->-----  
-- DEMINEUR --  
-- P_Tile --  
--  
-- AUTEUR : KAJI9 --  
-- DATE : 17/06/2013 --  
--  
-- Ce package gère les cases de la grille. Il définit les  
types--  
--T_Tile_Record et T_Tile ainsi que les programmes  
necessaires--  
--pour initialiser, modifier ou détruire une case. --  
--La variable globale Drapeaux_restants y est déclarée  
ainsi --  
--que le type T_Status indiquant l'état d'une case. --  
--  
-----  
  
WITH Gtk.Button ; USE Gtk.Button ;  
WITH Gtk.Image ; USE Gtk.Image ;  
WITH Gtk.Label ; USE Gtk.Label ;  
  
  
PACKAGE P_Tile IS  
-----  
-- VARIABLE GLOBALE --  
-----  
  
Drapeaux_Restants : Integer;  
--Permet le décompte des drapeaux utilisés et donc des bombes  
découvertes  
-----  
-- TYPES --  
-----  
  
TYPE T_Status IS (Normal, Flag, Dug);  
--Indique l'état d'une case :  
-- Normal : la case existe encore et ne porte pas de drapeau  
-- Flag : la case porte un drapeau  
-- Dug : la case a été creusée, le bouton n'existe plus  
-----  
TYPE T_Tile_Record IS TAGGED RECORD  
Btm : Gtk.Button;  
Img : Gtk.Image;  
Txt : Gtk.Label;  
Mine : Boolean := false;  
Nb : Integer := 0;  
Status : T_Status := Normal;  
END RECORD;  
-----  
TYPE T_Tile IS ACCESS ALL T_Tile_Record;  
--Les types permettant de manipuler les cases de la grille  
-- Btm, Img, Txt sont les widgets correspondants  
-- Mine indique si la case est minée  
-- Nb indique le nombre de bombes alentours  
-- Status indique l'état de la case  
-----  
-- PROGRAMMES --  
-----  
PROCEDURE Init_Tile (T : IN OUT T_Tile);  
--Initialise la case  
-----  
PROCEDURE Change_State(T : ACCESS T_Tile_Record'Class);  
--Change l'état d'une case de Normal à Flag ou inversement  
-----  
PROCEDURE Destroy (T : ACCESS T_Tile_Record'Class);  
-----
```

```
--Détruit le bouton de la case, change son statut et charge
l'image ou le texte à afficher

PRIVATE
  FUNCTION Set_Text (N : Integer) RETURN String;
    --Définit le texte à afficher sur une case ainsi que sa
    couleur,
    --N est le nombre à afficher
END P_Tile;
```

Voici maintenant le fichier définissant un tableau de cases :

Code : Ada - P_Tile.Tile_Array.ads

```
-- DEMINEUR --
-- P_Tile.Tile_Array --
-- --
-- AUTEUR : KAJI9 --
-- DATE : 17/06/2013 --
-- --
-- Ce package gère les tableaux de T_Tile (cf package
P_Tile) --
-- Il définit le type T_Tile_Array ainsi que les programmes
-- --
-- pour initialiser le tableau et pour tester si le joueur
a --
-- gagné. --
-- --

PACKAGE P_Tile.Tile_Array IS

  -----
  -- TYPE --
  -----

  TYPE T_Tile_Array IS ARRAY(integer range <>, integer range <>) OF
T_Tile;

  -----
  -- PROGRAMMES --
  -----

  PROCEDURE Init_Tile_Array(T : IN OUT T_Tile_Array;
Width,Height,Bombs : Integer);
  --Init_Tile_Array() permet de créer un tableau complet
  ainsi que de placer aléatoirement
  --des mines et d'affecter à chaque case le nombre de mines
  alentour.
  -- Width : largeur de la grille
  -- Height : hauteur de la grille
  -- Bombs : nombre de bombes

  FUNCTION Victory(T : IN T_Tile_Array) RETURN Boolean;
  --Victory() Renvoie TRUE si toutes les cases non minées ont
  été découverte, et
  --FALSE s'il reste des cases à creuser

PRIVATE
  PROCEDURE Increase(T : IN OUT T_Tile_Array ; X,Y : Integer);
  --Increase() permet d'augmenter le nombre de bombes connues
  d'une case
  --de 1 point. X et Y sont les coordonnées de la bombe.

END P_Tile.Tile_Array ;
```

Voici maintenant le fichier définissant le type T_Game. Il s'agit du package principal :

Code : Ada - Main_Window.ads

```
-- DEMINEUR --
-- Main_Window --
-- --
-- AUTEUR : KAJI9 --
-- DATE : 17/06/2013 --
-- --
-- Ce package définit les types T_Game_Record et T_Game qui
-- contiennent les informations liées à la partie, notamment
la --
-- grille de cases ou les principaux widgets. Il définit
aussi --
-- la fonction de callback (Click_on) et la procédure --
-- d'initialisation.
-- --

WITH Gtk.Window ;      USE Gtk.Window ;
WITH Gtk.Button ;      USE Gtk.Button ;
WITH Gtk.Table;        USE Gtk.Table;
WITH Gtk.Handlers ;   USE Gtk.Handlers ;
WITH Gdk.Events ;     USE Gdk.Events ;
WITH Gtk.Widget ;      USE Gtk.Widget ;
WITH Gtk.Box ;          USE Gtk.Box ;
WITH Gtk.Label ;        USE Gtk.Label ;
WITH P_Tile ;           USE P_Tile ;
WITH P_Tile.Tile_Array ; USE P_Tile.Tile_Array ;

PACKAGE Main_Window IS

  -----
  -- VARIABLE GLOBALE --
  -----

  Title : CONSTANT String := "Démineur";
  --Titre du jeu

  -----
  -- TYPES --
  -----

  TYPE T_Game_Record(width,Height,bombs : integer) IS RECORD
    Tab : T_Tile_Array(1..width, 1..height);
    X,Y : Integer;
    Win : Gtk.Window;
    Grille : Gtk_Table;
    Compteur : Gtk_Label;
    Box : Gtk_Vbox;
  END RECORD;

  TYPE T_Game IS ACCESS ALL T_Game_Record;
  --Contient la plupart des informations sur la partie :
  -- Width : largeur de la grille de cases
  -- Height : hauteur de la grille de cases
  -- Bombs : Nombre de bombes
  -- Tab : grille de cases
  -- X, Y : variables permettant de transmettre les coordonnées
  de la case cliquée
  -- Win : widget fenêtre du jeu
  -- Grille : widget GTK Table contenant tous les boutons
  -- Compteur : widget affichant la variable globale
  Drapeaux_restants
  -- Box : widget contenant Compteur et Box

  -----
  -- PACKAGES --
  -----

  PACKAGE P_Callback IS
    NEW
    Gtk.Handlers.User_Return_Callback(Gtk_Button_Record, boolean,
T_Game record);
    USE P_Callback;
    --Package des callback
```

```

-----  

-- PROGRAMMES --  

-----  

PROCEDURE Init_Game(Game : IN OUT T_Game ; Width,Height,Bombs : Integer) ;  

  --Procédure d'initialisation du jeu  

  --Les paramètres correspondent à ceux du type T_Game_Record  

FUNCTION click_on(Emetteur : ACCESS Gtk_Button_Record'Class ;  

  Event : Gdk_Event ;  

  Game : T_Game_Record) RETURN Boolean ;  

  --Callback appellé lorsque le joueur clique sur un bouton  

  --Cela-ci permet de placer ou d'enlever un drapeau  

  --mais aussi de creuser une case  

PRIVATE  

  PROCEDURE Init_Window (Game : T_Game) ;  

    --initialise la fenêtre de jeu  

  PROCEDURE Init_Compteur (Game : T_Game) ;  

    --initialise le compteur  

  PROCEDURE Init_Box (Game : T_Game) ;  

    --initialise le paramètre Box et y ajoute les widgets  

  Compteur Grille  

  PROCEDURE Init_Grille (Game : T_Game ; Width,Height,Bombs : Integer) ;  

    --initialise la grille de boutons et connecte à chacun son  

  callback  

  PROCEDURE Set_Compteur (Game : T_Game_Record) ;  

    --met à jour le compteur de drapeaux  

  PROCEDURE Explosion (Game : T_Game_Record ; X,Y : Integer) ;  

    --affiche la bombe et lance la boîte de dialogue de défaite  

  PROCEDURE Creuser_Autour(Game : T_Game_Record ; X,Y : Integer) ;  

    --détruit les 8 cases entourant la case de coordonnées (X,Y)  

    --appelle la procédure Creuser si l'une des 8 cases est  

  nulle  

  PROCEDURE Creuser (Game : T_Game_Record ; X,Y : Integer) ;  

    --détruit la case de coordonnées (X,Y). Lance explosion si la  

  case  

    --est minée ; lance Creuser_automatique si la case est nulle  

END Main_Window ;

```

Enfin, un package annexe permettant l'ouverture de boîte de dialogue pour la victoire ou la défaite :

```

Code : Ada - P_Dialog.ads  

-----  

-- DEMINEUR --  

-- P_Dialog --  

-- --  

-- AUTEUR : KAJI9 --  

-- DATE : 17/06/2013 --  

-- --  

-- Ce package définit les deux boîtes de dialogue du jeu :  

-- --Loose_Dialog qui s'ouvre si vous cliquez sur une mine et  

-- --Win_Dialog qui s'ouvre si vous avez découvert toutes les  

-- --mines. Il fournit également les procédures nécessaires à  

-- --initialisation --  

-----  

WITH Gtk.Message_Dialog ; USE Gtk.Message_Dialog ;  

WITH Gtk.Window ; USE Gtk.Window ;  

PACKAGE P_Dialog IS  

-----  

-- VARIABLES GLOBALES --  

-----  

Loose_Dialog : Gtk_Message_Dialog ;  

Win_Dialog : Gtk_Message_Dialog ;  

-----  

-- PROGRAMMES --  

-----  

PROCEDURE Init_Loose_Dialog(Parent : Gtk.Window) ;  

PROCEDURE Init_Win_Dialog (Parent : Gtk.Window) ;  

  --Initialisent les boîtes dialogues ci-dessus  

  -- Parent : indique la fenêtre mère (Game.Win)  

END P_Dialog ;

```

Le corps des packages

Si vous souhaitez comprendre le fonctionnement des programmes décrits plus haut, en voici le détail :

```

Code : Ada - P_Tile.adb  

-----  

-- DEMINEUR --  

-- P_Tile --  

-- --  

-- AUTEUR : KAJI9 --  

-- DATE : 17/06/2013 --  

-- --  

-- Ce package gère les cases de la grille. Il définit les  

types--  

  --T_Tile_Record et T_Tile ainsi que les programmes  

necessaires--  

  --pour initialiser, modifier ou détruire une case. --  

  --la variable globale Drapeaux_restants y est déclarée  

ainsi --  

  --que le type T_Status indiquant l'état d'une case. --  

-----  

PACKAGE BODY P_Tile IS  

  PROCEDURE Init_Tile(T : in out T_Tile) IS  

  BEGIN  

    T := new T_Tile_record ;  

    GTK.new(T.Btn) ;  

    T.Mine := False ;  

    T.NB := 0 ;  

    T.Status := normal ;  

  END Init_Tile ;  

  PROCEDURE Change_State(T : ACCESS T_Tile_Record'Class) IS  

  BEGIN  

    IF T.status = Normal  

      THEN T.Status := Flag ;  

        GTK.New(T.Img,"./drapeau-bleu.png") ;  

        T.Btn.Add(T.Img) ;  

        T.Img.Show ;  

        Drapeaux_restants := Drapeaux_restants - 1 ;  

    ELSE T.Status := Normal ;  

      T.Img.Destroy ;  

      Drapeaux_restants := Drapeaux_restants + 1 ;  

    END IF ;  

  END Change_State ;  

  FUNCTION Set_Text(N : Integer) RETURN String IS  

  BEGIN  

    CASE N IS  

      WHEN 1 => RETURN "<span font_desc='comic sans ms 12'  

foreground='blue'>1</span>" ;

```

```

    WHEN 2 => RETURN "<span font_desc='comic sans ms 12' foreground="#096A09"></span>" ;
    WHEN 3 => RETURN "<span font_desc='comic sans ms 12' foreground=><span>" ;
    WHEN 4 => RETURN "<span font_desc='comic sans ms 12' foreground="#003399"></span>" ;
    WHEN 5 => RETURN "<span font_desc='comic sans ms 12' foreground="#6C0277"></span>" ;
    WHEN 6 => RETURN "<span font_desc='comic sans ms 12' foreground="#87591A"></span>" ;
    WHEN 7 => RETURN "<span font_desc='comic sans ms 12' foreground="#D9E614"></span>" ;
    WHEN 8 => RETURN "<span font_desc='comic sans ms 12' foreground="#808000"></span>" ;
    WHEN OTHERS => RETURN "" ;
  END CASE ;
END Set_Text ;

```

Code : Ada - p_tile.tad

```

----- DEMINEUR -----
-- P_Tile.Tile_Array --
-- --
-- AUTEUR : KAJI9 --
-- DATE : 17/06/2013 --
-- --
-- Ce package gère les tableaux de T_Tile (cf package
P_Tile) --
-- Il définit le type T_Tile_Array ainsi que les programmes
-- pour initialiser le tableau et pour tester si le joueur
a -- --gagné. --
-----

WITH Ada.Numerics.Discrete_Random ;

PACKAGE BODY P_Tile.Tile_Array IS

  PROCEDURE Init_Tile_Array(T : IN OUT T_Tile_Array;
width,height,bombs : integer) IS
    subtype random_range is integer range 1..width*height ;
    PACKAGE P_Random IS
      NEW Ada.Numerics.Discrete_Random(Random_Range) ;
    USE P_Random ;
    G : Generator ;
    X,Y : Integer ;
    Reste : Integer := Bombs ;
  BEGIN
    Reset(G) ;
    --Création des cases
    FOR J IN 1..height LOOP
      FOR I IN 1..width LOOP
        Init_Tile(T(I,J)) ;
      END LOOP ;
    END LOOP ;
    --Placement aléatoire des bombes et calcul des nombres
associés à chaque case
    WHILE Reste > 0 LOOP
      X := Random(G) mod Width + 1 ;
      Y := Random(G) mod Height + 1 ;
      IF T(X,Y).Mine = false
      THEN T(X,Y).Mine:=True ;
        Increase(T,X,Y) ;
        Reste := Reste - 1 ;
      END IF ;
    END LOOP ;
    END Init_Tile_Array ;

  PROCEDURE Increase(T : IN OUT T_Tile_Array ; X,Y : Integer) IS
    xmin,xmax,ymin,ymax : integer ;
  BEGIN
    xmin := integer'max(1 , x-1) ;
    xmax := integer'min(x+1, T'last(1)) ;
    ymin := integer'max(1 , y-1) ;
    ymax := Integer'Min(Y+1, T'Last(2)) ;
    FOR J IN ymin..ymax LOOP
      FOR I IN xmin..xmax LOOP
        T(I,J).Nb := T(I,J).Nb + 1 ;
      END LOOP ;
    END LOOP ;
    END Increase ;
  END Increase ;

  FUNCTION Victory(T : IN T_Tile_Array) RETURN Boolean IS
    Nb_mines,Nb_cases : integer := 0 ;
  BEGIN
    --Décompte du nombre de mines et de cases non détruites
    FOR J IN T'RANGE(2) LOOP
      FOR I IN T'RANGE(1) LOOP
        IF T(I,J).Status = normal or t(i,j).status = flag
        THEN nb_cases := nb_cases + 1 ;
        END IF ;
        IF T(I,J).Mine
        THEN Nb_Mines := Nb_Mines + 1 ;
        END IF ;
      END LOOP ;
    END LOOP ;
    --Renvoie du résultat
    RETURN Nb_Mines = Nb_Cases ;
  END Victory ;
END P_Tile.Tile_Array ;

```

Code : Ada - main_window.adb

```

----- DEMINEUR -----
-- Main_Window --
-- --
-- AUTEUR : KAJI9 --
-- DATE : 17/06/2013 --
-- --
-- Ce package définit les types T_Game_Record et T_Game qui
-- contiennent les informations liées à la partie, notamment
la --
-- grille de cases ou les principaux widgets. Il définit
aussi --
-- la fonction de callback (Click_on) et la procédure --
-- d'initialisation.
-----

WITH Glib ; USE Glib ;
WITH Glib.Convert ; USE Glib.Convert ;
WITH P_Dialog ; USE P_Dialog ;
WITH Gtk.Dialog ; USE Gtk.Dialog ;

```

```

WITH Gtk.Main ;           USE Gtk.Main ;

PACKAGE BODY Main_Window IS
  PROCEDURE Init_Window(Game : T_Game) IS
    BEGIN
      Gtk_New(Game.Win) ;
      Game.Win.Set_Default_Size(400,400) ;
      Game.Win.Set_Title(Locale_To_Utf8(title)) ;
      IF Game.Win.Set_Icon_From_File("mine-noire.png")
        THEN NULL ;
      END IF ;
    END Init_Window ;

  PROCEDURE Init_Compteur(Game : T_Game) IS
    BEGIN
      Gtk_New(Game.Compteur,"<span font_desc='DS-Digital 45'>" &
Integer'Image(drapeaux_restants) & "</span>" ) ;
      Game.Compteur.Set_Use_Markup(True) ;
    END Init_Compteur ;

  PROCEDURE Init_Box(Game : T_Game) IS
    BEGIN
      Gtk_New_Vbox(Game.Box) ;
      Game.Box.Pack_Start(Game.Compteur, expand => false, fill =>
false) ;
      Game.Box.pack_start(Game.Grille,   expand => true, fill =>
true) ;
      Game.Win.Add(Game.Box) ;
    END Init_Box ;

  PROCEDURE Init_Grille(Game : T_Game ; width,height,bombs :
integer) IS
    BEGIN
      --Création de la GTK_Table et du T_Tile_Array
      Gtk_New(Game.Grille,
        Guint(Width),
        Guint(Height),
        True) ;
      Init_Tile_Array(Game.Tab,
        Width,
        Height,
        Bombs) ;
      --Implantation des différents boutons et connexion de
      --chacun avec son callback
      FOR J IN 1..height LOOP
        FOR I IN 1..width LOOP
          Game.Grille.Attach(Game.Tab(I,J).Btn,
            Guint(I)-1,Guint(I),
            Guint(J)-1,Guint(J)) ;
          Game.X := I ;
          Game.Y := J ;
          Connect(Game.Tab(I,J).Btn,
            Signal_Button_Press_Event,
            To_Marshaller(click_on'ACCESS),
            Game.all) ;
        END LOOP ;
      END LOOP ;
    END Init_Grille ;

  PROCEDURE Init_Game(Game : in out T_Game ; width,height,bombs :
integer) IS
    BEGIN
      Game := NEW T_Game_Record(width,height,bombs) ;
      Init_Window(Game) ;
      Init_Compteur(Game) ;
      Init_Grille(Game,Width,Height,Bombs) ;
      Init_Box(Game) ;
      Game.Win.show_all ;
    end Init_Game ;

  PROCEDURE Set_Compteur(Game : T_Game_Record) IS
    BEGIN
      IF Drapeaux_Restants < 0
        THEN Game.Compteur.Set_Label("<span foreground = 'red'
font_desc='DS-Digital 45'>" &
Integer'Image(Drapeaux_Restants) &
          "</span>" ) ;
      ELSE Game.Compteur.Set_Label("<span foreground = 'black'
font_desc='DS-Digital 45'>" &
Integer'Image(Drapeaux_Restants) &
          "</span>" ) ;
      END IF ;
    END Set_Compteur ;

  FUNCTION click_on(Emetteur : ACCESS Gtk_Button_Record'Class ;
    Evenement : Gdk_Event ;
    Game : T_Game_record) RETURN Boolean IS
    X : CONSTANT Integer := Game.X ;
    Y : CONSTANT Integer := Game.Y ;
    PRAGMA Unreferenced(Emetteur) ;
  BEGIN
    --Choix des procédures à lancer selon le bouton cliqué
    CASE Get_Button(Evenement) IS
      WHEN 1 => Creuser(Game,X,Y) ;
      WHEN 3 => Game.Tab(X,Y).Change_State ;
      WHEN OTHERS => NULL ;
    END CASE ;
    --Teste de victoire et lancement éventuels de la boite de
    dialogue
    --de victoire. Notez bien le "AND THEN"
    IF Victory(Game.Tab) AND THEN Win_Dialog.Run = Gtk_Response_Ok
      THEN Main_Quit ;
    END IF ;
    RETURN False ;
  END Click_On ;

  PROCEDURE Explosion(Game : T_Game_Record ; X,Y : Integer) IS
  BEGIN
    --Affichage de l'image de la bombe cliquée
    Game.Grille.Attach(Game.tab(x,y).Img,
      Guint(X)-1,
      Guint(X),
      Guint(Y)-1,
      Guint(Y)) ;
    --Ouverture de la boîte de dialogue de défaite
    IF Loose Dialog.Run = Gtk_Response_Ok
      THEN Main_Quit ;
    END IF ;
  END Explosion ;

  PROCEDURE Creuser_Autour(Game : T_Game_Record ; X,Y : Integer) IS
    Xmin,Xmax,Ymin,Ymax : Integer ;
    title : T_Tile ;
  BEGIN
    Xmin := integer'max(1 , -x-1) ;
    Xmax := integer'min(x+1 , Game.Tab'last(1)) ;
    Ymin := integer'max(1 , -y-1) ;
    Ymax := Integer'Min(Y+1 , Game.Tab'Last(2)) ;
    --parcourt des 9 cases autour de (X,Y)
    FOR J IN Ymin..Ymax LOOP
      FOR I IN Xmin..Xmax LOOP
        Tile := Game.Tab(I,J) ;
        --si la case porte un chiffre, elle est simplement
        détruite,
        --sinon, on lance un appel récursif via la procédure
        Creuser()
        IF Tile.status = Normal and Tile.nb > 0
          THEN Tile.Destroy ;
          Tile.Status := Dug ;
          Game.Grille.Attach(Tile.txt,
            Guint(I)-1,Guint(I),
            Guint(J)-1,Guint(J)) ;
        ELSIF Tile.Status = normal
          THEN Creuser(Game,I,J) ;
      END LOOP ;
    END LOOP ;
  END Creuser() ;

```

```

        END IF ;
    END LOOP ;
END LOOP ;
END Creuser_Autour ;

PROCEDURE Creuser(Game : T_Game_Record ; X,Y : Integer) IS
    tile : CONSTANT T_Tile := Game.tab(x,y) ;
BEGIN
    Tile.Destroy ;
    --Si la case est minée
    IF Tile.Status = Normal AND Tile.Mine
    THEN Explosion(Game,X,Y) ;
    --Si la case n'est pas minée ni creusée
    ELSIF Tile.Status = Normal
    THEN Tile.Status := Dug ;
        Game.Grille.Attach(Tile.txt,
                            Guint(X)-1,Guint(X),
                            Guint(Y)-1,Guint(Y)) ;
    --Si la case est nulle, on lance Creuser_around()
    IF Tile.Nb = 0
    THEN Creuser_around(Game,x,y) ;
    END IF ;
END IF ;
END Creuser ;
END Main_Window ;

```

Code : Ada - P_Dialog.adb

```

-----  

-- DEMINEUR --  

-- P_Dialog --  

-- --  

-- AUTEUR : KAJI9 --  

-- DATE : 17/06/2013 --  

-- --  

-- Ce package définit les deux boîtes de dialogue du jeu :  

-- --Loose_Dialog qui s'ouvre si vous cliquez sur une mine et  

-- --Win_Dialog qui s'ouvre si vous avez découvert toutes les  

-- --mines. Il fournit également les procédures nécessaires à  

-- --initialisation --  

-----  

WITH Glib.Convert ;           USE Glib.Convert ;
WITH Gtk.Dialog ;            USE Gtk.Dialog ;

PACKAGE BODY P_Dialog IS
    PROCEDURE Init_Loose_Dialog(Parent : Gtk.Window)  IS
    BEGIN
        Gtk_New(Loose_Dialog,
                Parent,
                Modal,
                Message_Warning,
                Buttons_Ok,
                Locale_To_Utf8("Vous avez sauté sur une mine !"));
        Loose_dialog.set_title("Perdu") ;
    END Init_Loose_Dialog ;
    PROCEDURE Init_Win_Dialog(Parent : Gtk.Window)  IS
    BEGIN
        Gtk_New(Win_Dialog,
                Parent,
                Modal,
                Message_Warning,
                Buttons_Ok,
                Locale_To_Utf8("Vous avez trouvé toutes les mines !"));
        Win_dialog.set_title("Victoire") ;
    END Init_Win_Dialog ;
END P_Dialog ;

```

La procédure principale

Voici enfin le cœur du jeu. Vous constaterez qu'il est très sommaire.

Code : Ada - Demmeur.adb

```

WITH Ada.Text_Io ;           USE Ada.Text_Io ;
WITH Ada.Integer_Text_IO ;   USE Ada.Integer_Text_IO ;

WITH Gtk.Main ;              USE Gtk.Main ;
WITH Main_Window ;          USE Main_Window ;
WITH P_Dialog ;              USE P_Dialog ;
WITH P_Tile ;                USE P_Tile ;

PROCEDURE Demineur IS
    Width : Integer ;
    Height : Integer ;
    Bombs : Integer ;
BEGIN
    Put_Line("Indiquez la largeur de la grille : ") ; Get(width) ;
    Put_Line("Indiquez la hauteur de la grille : ") ; Get(height) ;
    Put_Line("Indiquez le nombre de bombes : ") ; Get(Bombs) ;
    Drapeaux_restants := Bombs ;
    DECLARE
        Game : T_Game ;
    BEGIN
        Init ;
        Init_Game (Game,width,height,bombs) ;
        Init_Loose_Dialog(Game.Win) ;
        Init_Win_Dialog (Game.win) ;
        Main ;
    END ;
END Demineur ;

```

Ce TP s'achève mais comme vous l'aurez compris, nous le reprendrons bientôt pour le compléter, notamment par un menu et quelques options.

Pistes d'amélioration :

- ajout d'une fonctionnalité pour le clic gauche et droit simultané;
- ajout de la balise point d'interrogation;
- ajout d'un bouton smiley pour commencer une nouvelle partie;
- ajout d'un chronomètre;
- ajout de modes spéciaux : le démineur Windows propose par exemple un mode caché "passe-muraille".

Les conteneurs II

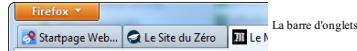
Nous avons déjà vu les conteneurs, pourquoi y revenir ?

Nous avons effectivement abordé les conteneurs principaux, les plus utiles. Ce chapitre est fait pour que nous abordions quelques conteneurs plus spécifiques de par leur apparence : ils ne sont pas seulement là pour organiser discrètement vos widgets ; ces conteneurs sont faits pour être vus ! Parmi eux, nous verrons les barres d'onglets, les barres de défilement, les panneaux, les cadres, les extensions et les boîtes détachables. Cela complétera votre boîte à outils, même si je serai pas aussi enthousiaste que pour le précédent chapitre (eh oui, il sera bientôt temps pour vous de voler de vos propres ailes et de lire le manuel ☺).

Les onglets

Fiche d'identité

Le premier conteneur de cette nouvelle série est donc l'onglet, ou plutôt la barre d'onglets. De quoi s'agit-il ? Eh bien c'est un widget que vous utilisez très certainement en ce moment même si vous lisez ce cours via internet :



La barre d'onglets

Eh oui, ce sont ces petites cases que vous ouvrez lorsque vous voulez visiter plusieurs sites web en même temps. Il est tout à fait possible d'en créer avec GTK, par exemple si vous souhaitez créer un navigateur internet ou un logiciel qui ouvrira tous ses fichiers dans une même fenêtre à l'aide de plusieurs onglets (c'est ce que fait par exemple le lecteur de fichier pdf Foxit Reader ou l'éditeur de code NotePad++). Voici donc la fiche d'identité des barres d'onglets :

- **Widget** : GTK_Notebook
- **Package** : GtkNotebook
- **Descendance** : GTK_Widget >> GTK_Container
- **Description** : Conteneur comportant plusieurs pages, chaque page permettant d'afficher un seul widget à la fois.

Méthodes

Créer des onglets

Le constructeur de GTK_Notebook est simplissime, je n'en parlerai donc pas. Vous devez toutefois savoir qu'une barre d'onglet GTK_Notebook, seule, ne fait pas grand chose et ne contiendra pas vos widgets. En fait, elle contient (ou pas) des GTK_Notebook_Page, c'est-à-dire des onglets ; et ce sont ces onglets qui contiendront un unique widget chacun. Il est donc important d'en créer et de les ajouter à votre barre. Tout cela se fait simplement avec les méthodes suivantes :

Code : Ada

```
procedure Append_Page(Notebook : access Gtk_Notebook_Record;
                      Child   : access Gtk_Widget_Record'Class;
                      Tab_Label : access Gtk_Widget_Record'Class);
procedure Append_Page(Notebook : access Gtk_Notebook_Record;
                      Child   : access Gtk_Widget_Record'Class);
procedure Append_Page_Menu(Notebook : access Gtk_Notebook_Record;
                           Child   : access
                           Gtk_Widget_Record'Class;
                           Tab_Label : access
                           Gtk_Widget_Record'Class;
                           Menu_Label : access
                           Gtk_Widget_Record'Class);
procedure Prepend_Page(Notebook : access Gtk_Notebook_Record;
                       Child   : access Gtk_Widget_Record'Class;
                       Tab_Label : access Gtk_Widget_Record'Class);
procedure Prepend_Page_Menu(Notebook : access Gtk_Notebook_Record;
                           Child   : access
                           Gtk_Widget_Record'Class;
                           Tab_Label : access
                           Gtk_Widget_Record'Class;
                           Menu_Label : access
                           Gtk_Widget_Record'Class);
procedure Insert_Page(Notebook : access Gtk_Notebook_Record;
                      Child   : access Gtk_Widget_Record'Class;
                      Tab_Label : access Gtk_Widget_Record'Class;
                      Position : Gint);
procedure Insert_Page_Menu(Notebook : access Gtk_Notebook_Record;
                           Child   : access
                           Gtk_Widget_Record'Class;
                           Tab_Label : access
                           Gtk_Widget_Record'Class;
                           Menu_Label : access
                           Gtk_Widget_Record'Class;
                           Position : Gint);
procedure Remove_Page(Notebook : access Gtk_Notebook_Record;
                      Page_Num : Gint);
```

Avec l'expérience acquise, vous devriez être capable de comprendre le sens de ces différentes méthodes sans mon aide. Toujours pas ? Rappelez vous : *Append* signifie Ajouter à la fin ; *Prepend* signifie Ajouter au début ; *Insert* signifie Insérer (à une position donnée) et *Remove* signifie Retirer, supprimer. Ces méthodes vont donc vous permettre d'ajouter, insérer ou supprimer des onglets (des GTK_Notebook_Page) à votre barre.

Que signifient les différents paramètres ? Prenons un exemple : la première méthode *Append_Page()*. Le paramètre Notebook est, aïe, encore besoin de le préciser, votre barre d'onglets. Le paramètre *Child* indique le widget que vous souhaitez ajouter. L'onglet sera automatiquement généré, pas besoin de constructeur pour les GTK_Notebook_Page. Bien entendu, ce widget peut être un conteneur complexe, contenant lui-même divers boutons, étiquettes, zones de saisie... Enfin, le troisième paramètre *Tab_Label* correspond au nom de l'onglet. Celui-ci n'est pas un *string* comme on aurait pu le croire mais un GTK_Label. Pourquoi ? Eh bien parce que le titre de votre onglet peut-être du texte (généralement un GTK_Label) mais aussi une GTK_Box afin d'afficher un image avec votre texte ! Si vous ne renseignez pas le paramètre *Tab_Label*, l'onglet sera nommé «Page 1», «Page 2»... par défaut.

Ah, d'accord ! Et je me doute que les paramètres *Position* correspondent à la position où l'on souhaite insérer un onglet. Par contre à quoi servent les paramètres *Menu_Label* ? ☺

Nous reviendrons sur ce paramètre un peu plus tard. Retenez qu'il s'agit d'un nom alternatif pour votre onglet et qui sera affiché lorsque l'utilisateur passera par le menu. Voyons maintenant les méthodes pour modifier ou accéder aux paramètres de base de vos onglets.

Modifier ou lire les paramètres de base des onglets

Si vous souhaitez obtenir ou modifier l'étiquette servant de titre à votre onglet, vous pourrez utiliser les méthodes suivantes :

Code : Ada

```
function Get_Tab_Label(Notebook : access Gtk_Notebook_Record;
                      Child   : access Gtk_Widget_Record'Class)
return Gtk_Widget;
procedure Set_Tab_Label(Notebook : access Gtk_Notebook_Record;
                      Child   : access Gtk_Widget_Record'Class;
                      Tab_Label : access Gtk_Widget_Record'Class);
```

Vous devez renseigner la barre d'onglets (Notebook) bien entendu, mais également le widget contenu dans l'onglet visé. À noter que si vous modifiez le titre (Tab_Label), vous devrez actualiser son affichage avec *show()* ou *show_all()*. Vous pouvez obtenir les mêmes résultats seulement en utilisant les *string* avec les méthodes suivantes :

Code : Ada

```
procedure Set_Tab_Label_Text(Notebook : access Gtk_Notebook_Record;
                            Child   : access
                            Gtk_Widget_Record'Class;
                            Tab_Text : UTF8_String);
function Get_Tab_Label_Text(Notebook : access Gtk_Notebook_Record;
                           Child   : access
                           Gtk_Widget_Record'Class)
                           return UTF8_String;
```

Vous pouvez également modifier un onglet via son numéro, sans connaître le widget qu'elle contient :

Code : Ada

```
procedure Set_Tab(Notebook : access Gtk_Notebook_Record;
                  Page_Num : Gint;
                  Tab_Label : access Gtk_Widget_Record'Class);
```

Enfin, vous retrouvez les mêmes méthodes pour modifier le paramètre `Menu_Label` de votre onglet :

Code : Ada

```
function Get_Menu_Label (Notebook : access Gtk_Notebook_Record;
                        Child : access Gtk_Widget_Record'Class)
return Gtk_Widget;
procedure Set_Menu_Label (Notebook : access Gtk_Notebook_Record;
                        Child : access Gtk_Widget_Record'Class;
                        Menu_Label : access Gtk_Widget_Record'Class);
procedure Set_Menu_Label_Text (Notebook : access Gtk_Notebook_Record;
                              Child : access Gtk_Widget_Record'Class;
                              Menu_Text : UTF8_String);
function Get_Menu_Label_Text (Notebook : access Gtk_Notebook_Record;
                             Child : access Gtk_Widget_Record'Class)
return UTF8_String;
```

Jouer avec les onglets

Ensuite, le nombre d'onglets pouvant augmenter ou diminuer au cours de l'utilisation de votre programme, il est important que vous puissiez connaître le nombre d'onglets ou lequel est actif. Ainsi pour connaître quel onglet est actif, utilisez la méthode `Get_Current_Page()` et si vous souhaitez changer d'onglet, utilisez la méthode `Set_Current_Page()` :

Code : Ada

```
function Get_Current_Page (Notebook : access Gtk_Notebook_Record)
return Gint;
procedure Set_Current_Page (Notebook : access Gtk_Notebook_Record;
                            Page_Num : Gint := -1);
--Si Page_Num vaut -1, cela renverra automatiquement à la dernière page.
```



La première page porte le numéro 0 et non 1 !

Vous pourrez obtenir le nombre total de pages grâce à `Get_N_Pages()` ainsi que le numéro correspondant à un widget avec `Page_Num()`

Code : Ada

```
function Get_N_Pages (Notebook : access Gtk_Notebook_Record) return
Gint;
function Page_Num (Widget : access Gtk_Notebook_Record'Class;
                   Child : access Gtk_Widget_Record'Class) return
Gint;
```

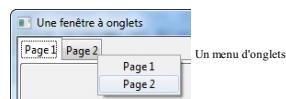
Maintenant, si vous souhaitez pouvoir changer d'onglet en passant au suivant (next) ou au précédent (previous) :

Code : Ada

```
procedure Next_Page (Notebook : access Gtk_Notebook_Record);
procedure Prev_Page (Notebook : access Gtk_Notebook_Record);
```

Modifier le comportement et le menu

Lorsque l'utilisateur a ouvert de très nombreux onglets, il peut être pratique d'accéder à n'importe quel onglet, non pas en cliquant dessus après l'avoir longuement cherché dans la liste mais plutôt en le sélectionnant via un menu accessible d'un simple clic droit :



Pour cela vous devrez autoriser (ou interdire) ces menus grâce aux méthodes suivantes :

Code : Ada

```
procedure Popup_Enable (Notebook : access Gtk_Notebook_Record); --
autoriser
procedure Popup_Disable (Notebook : access Gtk_Notebook_Record); --
interdire
```

Par défaut, le titre du menu est celui de l'onglet. Mais nous avons vu que vous pouviez modifier ce titre pour notamment indiquer l'action correspondante (`«enregistrez-vous»`, `«Lire les conditions d'utilisation»`...). Mais un autre problème se pose : lorsque l'utilisateur ouvrira beaucoup d'onglets, soit la fenêtre devra s'agrandir en conséquence, soit les onglets devront se rétrécir pour laisser de la place aux nouveaux venus. Pas très esthétique tout cela. Une solution pour pallier à ce problème est de ne pas tous les afficher et de permettre l'accès aux onglets cachés via des boutons :



Pour cela il suffit d'utiliser la méthode `Set_Scrollable()` en indiquant `TRUE` comme paramètre.

Code : Ada

```
procedure Set_Scrollable (Notebook : access Gtk_Notebook_Record;
                           Scrollable : Boolean := True);
function Get_Scrollable (Notebook : access Gtk_Notebook_Record)
return Boolean;
```

Déplacer les onglets

Pour organiser son travail, l'utilisateur souhaitera peut-être regrouper certains onglets entre eux. Si vous souhaitez qu'un onglet puisse être déplacé au sein d'une barre d'onglets, utilisez la méthode `Set_Tab_Reorderable()` :

Code : Ada

```
function Get_Tab_Reorderable (Notebook : access Gtk_Notebook_Record;
                               Child : access
Gtk_Widget_Record'Class);
```

```

    Position : Gint) return Boolean;
procedure Set_Tab_Reorderable(Notebook : access
Gtk_Notebook_Record; Child : access
Gtk_Widget_Record'Class; Reorderable : Boolean := True);

```

Mais il est possible de pousser le vice encore plus loin en permettant de déplacer un onglet d'une barre dans une autre, pour peu que ces deux barres fassent partie d'un même groupe. Voici les méthodes en jeu, je déroulerai un exemple tout de suite après :

Code : Ada

```

function Get_Tab_Detachable (Notebook : access Gtk_Notebook_Record;
Child : access
Gtk_Widget_Record'Class; Position : Gint) return Boolean;
procedure Set_Tab_Detachable(Notebook : access
Gtk_Notebook_Record; Child : access
Gtk_Widget_Record'Class; Detachable : Boolean := True);
procedure Set_Group_Id(Notebook : access Gtk_Notebook_Record;
Group_Id : Gint);
function Get_Group_Id (Notebook : access Gtk_Notebook_Record) return
Gint;

```

Vous devrez tout d'abord indiquer quel onglet sera détachable en indiquant le widget qu'il contient (`child`). Puis vous devrez lier les barres d'onglet entre elles en leur attribuant un même numéro de groupe avec `Set_Group_Id()`. Voici un exemple plus concrètement :

Code : Ada

```

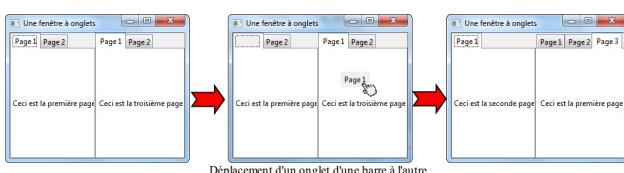
WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Label ; USE Gtk.Label ;
WITH Gtk.Notebook ; USE Gtk.Notebook ;
WITH Glib.Convert ; USE Glib.Convert ;
WITH Gtk.Box ; USE Gtk.Box ;

PROCEDURE MesOnglets IS
Win : Gtk.Window ;
Boite : Gtk_HBox ;
Onglets1, Onglets2 : Gtk_Notebook ;
Lbl1, Lbl2, Lbl3, Lbl4 : Gtk_Label ;

BEGIN
Init ;
--Création de la fenêtre
Gtk_New(Win) ;
Win.Set_Default_Size(150,200) ;
Win.Set_Title(locale_to_utf8("Une fenêtre à onglets")) ;
--Création d'une Horizontal Box pour séparer les deux barres
d'onglets
Gtk_new_hbox(boite) ;
Win.Add(Boite) ;
--Création des deux barres d'onglets
Gtk_New(Onglets1) ;
Gtk_New(Onglets2) ;
Boite.Pack_Start(Onglets1) ;
Boite.Pack_Start(Onglets2) ;
--Création de quatre étiquettes qui constitueront les quatre
onglets
Gtk_New(Lbl1, Locale_To_Utf8("Ceci est la première page")) ;
Gtk_New(Lbl2, Locale_To_Utf8("Ceci est la seconde page")) ;
Gtk_New(Lbl3, Locale_To_Utf8("Ceci est la troisième page")) ;
Gtk_New(Lbl4, Locale_To_Utf8("Ceci est la quatrième page")) ;
--Création des onglets de la barre n°1
Onglets1.Append_Page(Lbl1) ;
Onglets1.Append_Page(Lbl2) ;
Onglets1.Set_Tab_detachable(Lbl1) ;
Onglets1.Set_Tab_detachable(Lbl2) ;
--Création des onglets de la barre n°2
Onglets2.Append_Page(Lbl3) ;
Onglets2.Append_Page(Lbl4) ;
Onglets2.Set_Tab_detachable(Lbl3) ;
Onglets2.Set_Tab_detachable(Lbl4) ;
--Création d'un même groupe pour les deux barres d'onglet
Onglets1.Set_Group_Id(1) ;
Onglets2.Set_Group_Id(1) ;
Win.show_all ;
Main ;
END MesOnglets ;

```

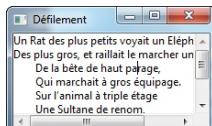
Le code ci-dessus nous donne la fenêtre sur la figure suivante. Remarquez que je n'ai pas donné de titre particulier à mes onglets et que GTK les renomme automatiquement lors d'un déplacement, ce qui crée un décalage entre le titre de l'onglet et son contenu.



Déplacement d'un onglet d'une barre à l'autre
Les barres de défilement
Fiche d'identité

Si vous avez déjà effectué quelques tests des divers widgets abordés depuis le début de cette partie, vous avez du vous rendre compte que selon le dimensionnement de la fenêtre, tout n'était pas toujours visible ou bien, inversement, que la fenêtre ne pouvait pas toujours être redimensionnée afin que certains widgets restent affichés. Par exemple, lorsque l'on utilise un éditeur de texte, la fenêtre ne se redimensionne pas quand on écrit trop de texte ; et pourtant, c'est bien ce qui se passe actuellement. Nous devons donc ajouter des barres de défilement horizontales et verticales à nos fenêtres pour pouvoir tout afficher.

- **Widget :** GTK_Scrolled_Window
- **Package :** Gtk.Scrolled_Window
- **Descendance :** GTK_Widget >> GTK.Container >> GTK_Bin
- **Description :** Ce widget permet d'afficher des barres de défilement horizontal en bas de la fenêtre et vertical à gauche de la fenêtre (voir la figure suivante)



Barres de défilement

Exemples d'utilisation

Utilisation avec un GTK_Text_View

La construction d'une GTK_Scrolled_Window se fait très simplement. Il vous suffit ensuite d'y ajouter avec la méthode Add() n'importe quel widget. Les barres de défilement ne sont donc rien de plus qu'un conteneur bordé de deux ascenseurs. Pour réaliser le programme que vous avez pu admirer ci-dessus, il n'a suffi d'écrire le code suivant :

Code : Ada

```
WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Scrolled_Window ; USE Gtk.Scrolled_Window ;
WITH Gtk.Text_View ; USE Gtk.Text_View ;
WITH Glib.Convert ; USE Glib.Convert ;

PROCEDURE Ascenseur IS
  Win : GTK_Window ;
  Defilement : GTK_Scrolled_Window ;
  Texte : GTK_Text_View ;
BEGIN
  Init ;
  --Création de la fenêtre principale
  Gtk_New(Win) ;
  Win.Set_Default_Size(200,100) ;
  Win.Set_Title(Locale_To_Utf8("Défilement")) ;
  --Création des barres de défilement
 Gtk_New(Defilement) ;
  Win.Add(Defilement) ;
  --Création de la zone de texte et ajout aux barres de
défilement
  Gtk_New(Texte) ;
  Defilement.Add(Texte) ;
  Win.Show_All ;
  Main ;
END Ascenseur ;
```

Utilisation avec un GTK_Fixed

Un problème se pose toutefois avec certains widgets. Tous ne supportent pas aussi bien les barres de défilement (aussi appelés ascenseurs). Les GTK_Text_View sont dits scrollable en langage GTK, c'est-à-dire qu'ils supportent très bien les ascenseurs. Mais les GTK_Fixed, par exemple, ne les supportent pas et sont dits non scrollables.

Pour mieux comprendre, faisons un petit exercice : vous allez réaliser un programme qui affiche une carte. Celle-ci est composée d'images simples : des cases vertes pour la terre, des cases bleues pour la mer. Celles-ci seront placées sur un GTK_Fixed auquel nous adjointrons des ascenseurs pour pouvoir garder l'accès à toutes les cases même lorsque l'on redimensionnera la fenêtre. Comme pour le démonter, je vous conseille de créer un tableau de widget. Voici une solution :

Secret (cliquez pour afficher)

Code : Ada

```
WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Scrolled_Window ; USE Gtk.Scrolled_Window ;
WITH Gtk.Image ; USE Gtk.Image ;
WITH Gtk.Fixed ; USE Gtk.Fixed ;
WITH Glib.Convert ; USE Glib.Convert ;
WITH Glib ; USE Glib ;

PROCEDURE Carte IS
  Win : GTK_Window ;
  Defilement : GTK_Scrolled_Window ;
  Fond : GTK_Fixed ;
  --Création d'un type tableau pour savoir si les cases sont
terrestres ou marines
  TYPE T_Array IS ARRAY(1..6,1..6) OF Boolean;
  Terre : CONSTANT T_Array :=
    ((True,True,True,False,False),
     (True,True,True,False,False),
     (True,False,False,False,True),
     (False,False,False,False,True),
     (True,False,False,False,False),
     (True,True,True,False,True));
  --Création d'un tableau contenant les images
  TYPE T_Image_Array IS ARRAY(1..6,1..6) OF GTK_Image ;
  Image : T_Image_Array ;

BEGIN
  Init ;
  --Création de la fenêtre
  Gtk_New(Win) ;
  Win.Set_Default_Size(200,100) ;
  Win.Set_Title(Locale_To_Utf8("Défilement")) ;
  --Création des ascenseurs
  Gtk_New(Defilement) ;
  Win.Add(Defilement) ;
  --Création du GTK_Fixed
  Gtk_New(Fond) ;
  Defilement.Add(Fond) ;
  --Création et ajout des images (chaque image a une taille
de 50 par 50 pixels
  FOR J IN 1..6 LOOP
    FOR I IN 1..6 LOOP
      IF Terre(I,J) THEN
        Image(I,J) := Gtk_New(Image(I,J),"terre.png") ;
      ELSE
        Image(I,J) := Gtk_New(Image(I,J),"mer.png") ;
      END IF ;
      fond.Put(Image(i,j),Gint((j-1)*50),Gint((i-1)*50)) ;
    END LOOP ;
  END LOOP ;
  Win.Show_All ;
  Main ;
END Carte ;
```

Malheureusement, le résultat n'est pas à la hauteur de nos attentes : les barres de défilement sont inutilisables et nous obtenons en sus une jolie erreur (voir la figure suivante) :



Gtk-WARNING**: gtk_scrolled_window_add(): cannot add non scrollable widget use
gtk_scrolled_window_add_with_viewport() instead



Un résultat décevant

Nous allons devoir utiliser un GTK_Viewport, un conteneur dont le seul but est de jouer les intermédiaires entre notre GTK_Fixed et notre GTK_Scrolled_Window. Deux solutions s'offrent à nous :

1. Créer nous-même un GTK_Viewport.
2. Utiliser une méthode en remplacement de Add() qui créera ce GTK_Viewport pour nous.

Dans le premier cas, sachez que le package concerné est (rien d'original me direz-vous) GTK.Viewport. Le constructeur est

simpissime, vous vous en tirez tout seul. Une fois vos widgets initialisés, vous devrez ajouter le GTK_Fixed à votre GTK_Vviewport qui à son tour devra être ajouté à votre GTK_Scrolled_Window. Long et fastidieux, non ?

Pour aller plus vite, nous opterons pour la seconde méthode en utilisant Add_With_Vviewport(). Le GTK_Vviewport est ainsi généré automatiquement sans encombrer votre code. Notre programme donne ainsi :

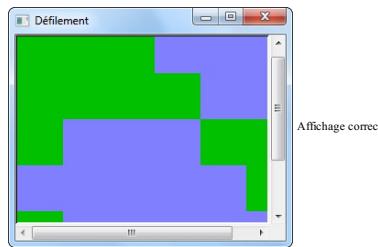
Code : Ada

```

WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Scrolled_Window ; USE Gtk.Scrolled_Window ;
WITH Gtk.Image ; USE Gtk.Image ;
WITH Gtk.Fixed ; USE Gtk.Fixed ;
WITH Glib.Convert ; USE Glib.Convert ;
WITH Glib ; USE Glib ;

PROCEDURE Carte IS
  Win : GTK_Window ;
  Defilement : GTK_Scrolled_Window ;
  Fond : GTK_Fixed ;
  TYPE T_Array IS ARRAY(1..6,1..6) OF Boolean;
  Terre : CONSTANT T_Array :=
    ((True,True,True,False,False),
     (True,True,True,False,False),
     (True,False,False,False,True),
     (False,False,False,False,False),
     (True,False,False,False,False),
     (True,True,True,False,True));
  TYPE T_Image_Array IS ARRAY(1..6,1..6) OF GTK_Image ;
  Image : T_Image_Array ;
BEGIN
  Init ;
  Gtk_New(Win) ;
  Win.Set_Default_Size(200,100) ;
  Win.Set_Title(Locale.To_Utf8("Défilement")) ;
  Gtk_New(Defilement) ;
  Win.Add(Defilement) ;
  Gtk_New(Fond) ;
  Defilement.Add_With_Vviewport(Fond) ;
  FOR J IN 1..6 LOOP
    FOR I IN 1..6 LOOP
      IF Terre(I,J) THEN
        Gtk_New(Image(I,J),"terre.png") ;
      ELSE
        Gtk_New(Image(I,J),"mer.png") ;
      END IF ;
      fond.Put((Image(I,J),Gint((j-1)*50),Gint((i-1)*50)) ;
    END LOOP ;
  END LOOP ;
  Win.Show_All ;
  Main ;
END Carte ;

```



Affichage correct

Méthodes

Voici quelques méthodes supplémentaires. Tout d'abord, si vous souhaitez obtenir l'une des deux barres de défilement :

Code : Ada

```

function Get_HScrollbar(Scrolled_Window :          access
  Gtk_Scrolled_Window_Record) :          access
  Gtk_Scrollbar;
function Get_VScrollbar(Scrolled_Window :          access
  Gtk_Scrolled_Window_Record) :          access
  Gtk_Scrollbar;

```

Chacune de ces méthodes renvoie une Gtk_Scrollbar, c'est-à-dire le widget servant de barre de défilement (je vous laisse regarder son package si vous souhaitez en savoir plus). Évidemment, Get_HScrollbar() renvoie la barre horizontale et Get_VScrollbar() la barre verticale. Enfin, voici quelques dernières méthodes utiles :

Code : Ada

```

procedure Set_Policy(Scrolled_Window :          access
  Gtk_Scrolled_Window_Record);
  H_Scrollbar_Policy : Gtk_Policy_Type;
  V_Scrollbar_Policy : Gtk_Policy_Type;
procedure Get_Policy(Scrolled_Window :          access
  Gtk_Scrolled_Window_Record);
  H_Scrollbar_Policy : out Gtk_Policy_Type;
  V_Scrollbar_Policy : out Gtk_Policy_Type;

procedure Set_Placement(Scrolled_Window :          access
  Gtk_Scrolled_Window_Record);
  Window_Placement : Gtk_Corner_Type;
function Get_Placement(Scrolled_Window :          access
  Gtk_Scrolled_Window_Record) return Gtk_Corner_Type;

```

Set_Policy() permet de gérer l'affichage des barres. Pour l'heure, celles-ci s'affichent toujours car les paramètres H_Scrollbar_Policy et V_Scrollbar_Policy ont leur valeur par défaut : Policy_Always. En fouillant dans le package Gtk.Enums vous trouverez les autres valeurs :

- Policy_Always : les barres sont toujours affichées.
- Policy_Automatic : les barres ne s'affichent que si nécessaire.
- Policy_Never : les barres ne sont jamais affichées.

Bien entendu, chaque barre peut avoir une politique d'affichage différente, comme toujours afficher la barre verticale mais jamais la barre horizontale. La méthode Set_Placement() indique l'emplacement des deux ascenseurs ou plus exactement, il indique dans quel coin de la fenêtre les ascenseurs ne sont pas présents. Par défaut, la barre verticale est à droite et la barre horizontale en bas. Le seul coin qui ne touche pas les ascenseurs est donc le coin supérieur gauche. Par défaut, le paramètre Window_Placement vaut donc Corner_Top_Left. Les autres valeurs sont : Corner_Top_Right, Corner_Bottom_Left et Corner_Bottom_Right.

Les panneaux

Fiche d'identité

- Widget : GTK_Paned
- Package : Gtk_Paned
- Descendance : GTK_Widget >> GTK_Container
- Description : Ce widget est une sorte de boîte, horizontale ou verticale, séparant deux widgets. À la différence des boîtes classiques, les panneaux peuvent être redimensionnés par l'utilisateur, à la façon des explorateurs de fichiers (voir la figure suivante).



Méthodes

Comme pour les boîtes, il existe deux sortes de GTK_Paned, les GTK_HPaned (panneau horizontal comme ci-dessus) et les GTK_VPaned (panneaux verticaux). Chacun dispose de son propre constructeur :GTK_New_HPaned() et GTK_New_VPaned().

Pour ajouter des widgets à vos panneaux, deux méthodes existent :Add1() et Add2(). La première ajoute le widget à gauche pour un panneau horizontal ou en haut pour un panneau vertical. Le second ajoutera donc le widget à droite ou en bas :

Code : Ada

```
procedure Add1(Paned : access Gtk_Paned_Record;
              Child : access Gtk_Widget_Record'Class);
procedure Add2(Paned : access Gtk_Paned_Record;
              Child : access Gtk_Widget_Record'Class);
```

Vous pouvez également paramétriser la répartition des widgets au sein du panneau avec les méthodes ci-dessous :

Code : Ada

```
procedure Pack1(Paned : access Gtk_Paned_Record;
                Child : access Gtk_Widget_Record'Class;
                Resize : Boolean := False;
                Shrink : Boolean := True);
procedure Pack2(Paned : access Gtk_Paned_Record;
                Child : access Gtk_Widget_Record'Class;
                Resize : Boolean := False;
                Shrink : Boolean := False);
```

L'attribut `Resize` indique si l'emplacement réservé au widget peut être redimensionné pour qu'il occupe le plus de place possible au démarrage. Bien sûr, cela n'est visible que si les widgets de gauche et de droite ne sont pas paramétrés de la même façon. L'attribut `Shrink` indique s'il est possible que l'utilisateur puisse redimensionner le panneau au point que le widget soit partiellement ou entièrement caché. Il est également possible de récupérer ces widgets avec les méthodes `Get_Child1()` et `Get_Child2()` :

Code : Ada

```
function Get_Child1(Paned : access Gtk_Paned_Record) return
    Gtk_Widget;
function Get_Child2(Paned : access Gtk_Paned_Record) return
    Gtk_Widget;
```

Enfin, dernière méthode. Si vous souhaitez positionner vous-même la séparation entre les deux widgets, utilisez la méthode `Set_Position()` en indiquant la distance voulue en pixels :

Code : Ada

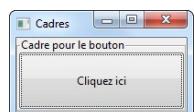
```
function Get_Position (Paned : access Gtk_Paned_Record) return
    Gint;
procedure Set_Position (Paned : access Gtk_Paned_Record; Position :
    Gint);
```

Les cadres

Les cadres simples

Fiche d'identité

- Widget : GTK_Frame
- Package : GTK.Frame
- Descendance : GTK_Widget >> GTK.Container >> GTK_Bin
- Description : C'est un conteneur pour un seul widget. Son but est en général de grouper de façon esthétique plusieurs widgets. De plus, il peut être doté d'un titre (voir la figure suivante).



Méthodes

Nous aurons très vite effectué le tour des méthodes car ce conteneur est simple d'utilisation. Commençons par le constructeur qui comporte un paramètre supplémentaire pour le titre du cadre (dans l'exemple ci-dessus, ce titre est "Cadre pour le bouton"). Notez que le titre est optionnel :

Code : Ada

```
procedureGtk_New (Frame : out Gtk_Frame; Label : UTF8_String :=
    "");
```

Vous disposez également de deux méthodes pour lire ou (re)définir directement le titre (`Get_Label()` et `Set_Label()`) mais aussi pour éditer l'étiquette elle-même (`Get_Label_Widget()` et `Set_Label_Widget()`) :

Code : Ada

```
function Get_Label (Frame : access Gtk_Frame_Record) return
    UTF8_String;
procedure Set_Label (Frame : access Gtk_Frame_Record;
                    Label : UTF8_String);
function Get_Label_Widget (Frame : access Gtk_Frame_Record) return
    Gtk_Widget;
procedure Set_Label_Widget (Frame : access Gtk_Frame_Record;
                           Label_Widget : access
                               Gtk_Widget_Record'Class);
```

Enfin, vous pouvez obtenir ou modifier l'alignement du titre avec les méthodes suivantes :

Code : Ada

```
procedure Get_Label_Align(Frame : access Gtk_Frame_Record;
                           Xalign : out Gfloat;
                           Yalign : out Gfloat);
```

```
procedure Set_Label_Align(Frame : access Gtk_Frame_Record;
                           Xalign : Gfloat;
                           Yalign : Gfloat);
```

Les paramètres Xalign et Yalign définissent bien entendu les alignements horizontaux et verticaux mais surtout, ont des valeurs comprises entre 0.0 et 1.0. Pour Xalign, la valeur 0.0 indique un titre aligné à gauche et 1.0 à droite. Une valeur de 0.25 indiquerait que le titre est placé à un quart de la longueur. Pour Yalign, 0.0 indiquerait que la bordure du cadre sera en bas par rapport au titre, une valeur de 1.0 indiquerait une bordure en haut du titre. Généralement, cette valeur est laissée à 0.5 pour obtenir une bordure partant du «milieu du titre».

Les cadres d'aspect

Fiche d'identité

- **Widget** : GTK_Aspect_Frame
- **Package** : GTK_Aspect_Frame
- **Descendance** : GTK_Widget >> GTK_Container >> GTK_Bin
- **Description** : Ce conteneur dérive des GTK_Frame et a donc les mêmes attributs. Il permet toutefois de placer de façon harmonieuse vos widgets en plaçant des espaces vides tout autour.

Méthodes

Le constructeur est ici la seule méthode vraiment complexe, il mérite donc que l'on s'y arrête :

Code : Ada

```
procedure Gtk_New (Aspect_Frame : out Gtk_Aspect_Frame;
                  Label      : UTF8_String := "";
                  Xalign    : Gfloat;
                  Yalign    : Gfloat;
                  Ratio     : Gfloat;
                  Obey_Child : Boolean);
```

Les paramètres Xalign et Yalign indiquent ici non plus l'alignement du titre mais celui du cadre tout entier. Imaginons que votre fenêtre soit bien plus large que nécessaire, si Xalign vaut 0.0, votre cadre s'alignera alors à gauche. S'il vaut 1.0, il s'alignera à droite. Et ainsi de suite, le fonctionnement est similaire à ce que nous venons de voir.

Le paramètre Ratio est un peu plus complexe à comprendre. Il s'agit du rapport *largeur / hauteur* du widget contenu dans votre cadre. Si vous placez un bouton dans votre cadre avec un ratio de 0.5, cela indiquerait que la largeur du bouton doit valoir la moitié de sa hauteur. Avec un ratio de 2.0, c'est l'inverse : la largeur est deux fois plus grande que la hauteur.



Alors dans ce cas, le dernier paramètre devrait vous satisfaire : si Obey_Child vaut TRUE, alors le ratio sera ignoré et le cadre devra «obéir au fils», c'est-à-dire prendre les dimensions souhaitées par le widget qu'il contient. Enfin, si vous souhaitez connaître ces paramètres ou les modifier, voici ce dont vous aurez besoin :

Code : Ada

```
procedure Set(Aspect_Frame : access Gtk_Aspect_Frame_Record;
              Xalign    : Gfloat;
              Yalign    : Gfloat;
              Ratio     : Gfloat;
              Obey_Child : Boolean);
function Get_Xalign(Aspect_Frame : access Gtk_Aspect_Frame_Record)
return Gfloat;
function Get_Yalign(Aspect_Frame : access Gtk_Aspect_Frame_Record)
return Gfloat;
function Get_Ratio (Aspect_Frame : access Gtk_Aspect_Frame_Record)
return Gfloat;
```

Les extenseurs

Fiche d'identité

- **Widget** : GTK_Expander
- **Package** : GTK_Expander
- **Descendance** : GTK_Widget >> GTK_Container >> GTK_Bin
- **Description** : C'est un conteneur pour un seul widget. Son but est d'afficher ou de masquer le widget qu'il contient (voir la figure suivante).



Méthodes

Comme pour les cadres simples, le constructeur des extenseurs exige un titre. Il vous est possible également de souligner certaines lettres comme pour les étiquettes avec GTK_New_With_Mnemonic() :

Code : Ada

```
procedure Gtk_New (Expander : out Gtk_Expander;
                  Label      : UTF8_String);
procedure Gtk_New_With_Mnemonic(Expander : out Gtk_Expander;
                               Label      : UTF8_String);
```

On retrouve également de nombreuses méthodes déjà vues avec les étiquettes ou les boutons, comme Set_Use_Markup() et Set_Use_Underline(). Je ne réexplique pas leur fonctionnement, mais je vous invite à relire les chapitres 2 et 5 de la partie Vsivous avez tout oublié.

Code : Ada

```
function Get_Use_Markup(Expander : access Gtk_Expander_Record)
return Boolean;
procedure Set_Use_Markup(Expander : access Gtk_Expander_Record;
                        Use_Markup : Boolean);
function Get_Use_Underline(Expander : access Gtk_Expander_Record)
return Boolean;
procedure Set_Use_Underline(Expander : access
                           Gtk_Expander_Record;
                           Use_Underline : Boolean);
function Get_Label_Fill(Expander : access Gtk_Expander_Record)
return Boolean;
procedure Set_Label_Fill(Expander : access Gtk_Expander_Record;
                        Label_Fill : Boolean);
function Get_Label(Expander : access Gtk_Expander_Record) return
UTF8_String;
procedure Set_Label(Expander : access Gtk_Expander_Record;
                   Label : UTF8_String);
function Get_Label_Widget(Expander : access Gtk_Expander_Record)
return Gtk_Widget;
procedure Set_Label_Widget(Expander : access
                           Gtk_Expander_Record;
                           Label_Widget : access
                           Gtk_Widget_Record'Class);
```

La méthode Set_Label_Fill() vous permet d'indiquer si vous souhaitez que le titre «remplisse» tout l'espace horizontal qui lui est attribué. Plus simplement, si le paramètre Label_Fill vaut TRUE, votre titre apparaîtra centré et non aligné à

gauche. Enfin, vous pouvez obtenir ou modifier le titre soit sous forme de String (avec `Get_Label()` et `Set_Label()`), soit sous forme de widget (avec `Get_Label_Widget()` et `Set_Label_Widget()`).

Rien de bien complexe dans tout cela. Voici maintenant les dernières méthodes (eh oui, ce conteneur est simple d'emploi) :

Code : Ada

```
function Get_Spacing (Expander : access Gtk_Expander_Record) return
Gint;
procedure Set_Spacing(Expander : access Gtk_Expander_Record;
Spacing : Gint);

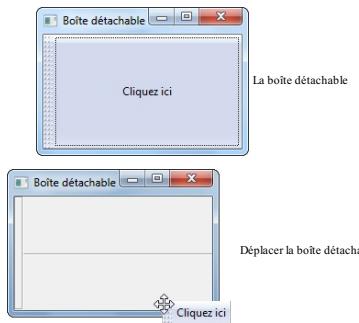
function Get_Expanded(Expander : access Gtk_Expander_Record) return
Boolean;
procedure Set_Expanded(Expander : access Gtk_Expander_Record;
Expanded : Boolean);
```

La méthode `Set_Spacing()` permet de spécifier le nombre de pixels séparant le titre et le widget fils lorsque l'extenseur est déroulé. Quand à `Set_Expanded()`, il permet d'indiquer si l'extenseur est déroulé (`TRUE`) ou pas (`FALSE`).

Les boîtes détachables

Fiche d'identité

- **Widget** : GTK_Handle_Box
- **Package** : GTK.Handle_Box
- **Descendance** : GTK_Widget >> GTK.Container >> GTK_Bin
- **Description** : Ce conteneur est conçu pour un unique widget. Mais il est possible de le «détacher» de la fenêtre, à la façon de certaines barres d'outils (voir la figure suivante).



Méthodes

Malgré ce que l'on pourrait croire, les boîtes détachables sont très simples à créer (mais peut-être pas toujours à utiliser). Je ne m'attarde pas sur le constructeur et j'en viens tout de suite aux trois méthodes utiles (oui c'est peu mais je n'y peux rien) :

Code : Ada

```
procedure Set_Handle_Position(Handle_Box : access
Gtk_Handle_Box_Record;
Position : Gtk_Position_Type);
function Get_Handle_Position(Handle_Box : access
Gtk_Handle_Box_Record) return Gtk_Position_Type;

procedure Set_Snap_Edge(Handle_Box : access Gtk_Handle_Box_Record;
Edge : Gtk_Position_Type);
function Get_Snap_Edge(Handle_Box : access Gtk_Handle_Box_Record)
return Gtk_Position_Type;

function Get_Child_Detached(Handle_Box : access
Gtk_Handle_Box_Record) return Boolean;
```

Tout d'abord, la méthode `Set_Handle_Position()` permet de positionner la zone de capture (vous savez, cette barre qui ressemble à une tôle gauffrée et qui permet de saisir le contenu pour le déplacer). Par défaut, cette zone de capture est placée à gauche, mais vous pouvez bien entendu modifier cette valeur en jetant un œil au package `Gtk.Enums`. Les valeurs possibles sont :

- `Pos_Left` : Zone de saisie à gauche.
- `Pos_Right` : Zone de saisie à droite.
- `Pos_Top` : Zone de saisie en haut.
- `Pos_Bottom` : Zone de saisie en bas.

Vous devez également savoir que la `GTK_Handle_Box` ne bouge pas. Lorsque vous croyez la détacher, en réalité, GTK crée une sorte de conteneur fantôme pour vous donner l'illusion du mouvement. Mais votre `GTK_Handle_Box` est toujours en place ! Mais lorsque vous voudrez remplacer le fantôme dans la `GTK_Handle_Box`, où devrez-vous le lâcher ? Il faut en fait faire coïncider un bord du fantôme avec un bord de la `GTK_Handle_Box`. Oui, mais lequel ? Pour le spécifier, utiliser la méthode `Set_Snap_Edge()`.

Enfin, la dernière méthode (`Get_Child_Detached()`), vous permettra de savoir si le widget a été détaché ou pas.

En résumé :

- Ces conteneurs ne peuvent afficher qu'un seul widget (à l'exception du panneau qui peut en afficher deux), vous devrez donc combiner ces conteneurs avec ceux vus précédemment (conteneurs à positions fixes, boîtes, tables ...).
- Ces conteneurs ont un intérêt principalement esthétique. Leur rôle est minime en ce qui concerne l'organisation des widgets dans votre fenêtre, mais énorme pour ce qui est de sa lisibilité. Ne les négligez pas.

Les widgets III : barres et menus

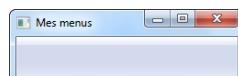
Reprenez notre chemin dans la forêt des widgets. Nous ne pouvons pas tous les voir mais je sais que vous attendez avec impatience de pouvoir ajouter des menus ou des barres d'icônes à vos programmes. C'est donc le cœur de ce troisième et dernier chapitre sur les widgets. Cela ne signifie pas que nous aurons tout vu, loin de là, mais nous aurons alors effectué un large tour des possibilités offertes par GTK.

La barre de menu

Créer une barre de menu

Fiche d'identité

- **Widget :** GTK_Menu_Bar
- **Package :** GtkMenu_Bar et Gtk.Menu_Shell
- **Descendance :** GTK_Widget >> GTK_Container >> GTK_Menu_Shell
- **Description :** La GTK_Menu_Bar est simplement la barre dans laquelle nous placerons nos divers menus. Sur la figure suivante, vous pouvez observer une barre de menu un peu vide et... sans menus.



La barre des menus



Le package GTK.Menu_Bar ne contient que très peu de méthodes ; je vous invite donc à jeter un œil au package GTK.Menu_Shell dont hérite GTK.Menu_Bar.

Quelques méthodes et un exemple

Le constructeur des GTK_Menu_Bar est simplissime et ne demande aucun paramètre. La seule méthode définie spécifiquement pour les GTK_Menu_Bar est la suivante :

Code : Ada

```
procedure Set_Pack_Direction(MenuBar : access Gtk_Menu_Bar_Record;
                             Pack_Dir : Gtk.Enums.Gtk_Pack_Direction);
function Get_Pack_Direction (MenuBar : access Gtk_Menu_Bar_Record)
                           return Gtk.Enums.Gtk_Pack_Direction;
```

C'est elle qui indiquera comment seront classés vos menus : de gauche à droite, de haut en bas... Les valeurs possibles pour le paramètre Pack_Dir sont :

- Pack_Direction_LTR : classement de gauche à droite (Left To Right)
- Pack_Direction_RTL : classement de droite à gauche (Right To Left)
- Pack_Direction_TTB : classement de haut en bas (Top To Bottom)
- Pack_Direction_BTT : classement de bas en haut (Bottom To Top)

Avec ceci nous pouvons dès et déjà créer une fenêtre contenant une barre de menu :

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;        USE Gtk.Window ;
WITH Gtk.Menu_Bar ;      USE Gtk.Menu_Bar ;
WITH Gtk.Enums ;          USE Gtk.Enums ;

PROCEDURE MesMenus IS
  Win          : Gtk.Window ;
  Barre        : Gtk.Menu_Bar ;
BEGIN
  Init ;
  --Création de la fenêtre
  Gtk.New(Win) ;
  Win.Set_Default_Size(250,25) ;
  Win.Set_Title("Mes menus") ;
  --Création de la barre de menu
 Gtk_New(Barre) ;
  Barre.set_pack_direction(Pack_Direction_LTR) ;   --cette ligne
  est superflue car par défaut,                      --GTK oriente
  le menu de gauche à droite
  Win.Add(Barre) ;
  --Finalisation
  Win.Show_All ;
  Main ;
END MesMenus ;
```

Nous obtenons ainsi la barre de menu vide présentée en introduction. Pour plus de méthodes, il faudra fouiller le package GTK.Menu_Shell. Voici quelques-unes qui nous serviront :

Code : Ada

```
procedure Append (Menu_Shell : access Gtk_Menu_Shell_Record;
                  Child    : access Gtk_Menu_Item_Record'Class);
procedure Prepend(Menu_Shell : access Gtk_Menu_Shell_Record;
                  Child    : access Gtk_Menu_Item_Record'Class);
procedure Insert (Menu_Shell : access Gtk_Menu_Shell_Record;
                  Child    : access Gtk_Menu_Item_Record'Class;
                  Position  : Gint);
```

Voilà des noms qui devraient vous paraître désormais : Append() ajoute à la fin, Prepend() ajoute au début et Insert() ajoute à la position indiquée. Où mais qu'ajoutent-elles ces méthodes ? Si vous avez observé leurs spécifications, vous avez du remarquer qu'elles permettent d'ajouter des GTK_Menu_Item.



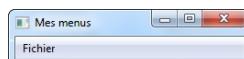
Qu'est-ce que c'est que ça encore ? 😊 Et puis j'aimerais que ma barre ne soit plus vide svp.

Ça tombe bien, car les GTK_Menu_Item sont justement les widgets qui vont remplir notre barre.

Créer et ajouter des items

Fiche d'identité

- **Widget :** GTK_Menu_Item
- **Package :** GtkMenu_Item
- **Descendance :** GTK_Widget >> GTK_Container >> GTK_Bin >> GTK_Item
- **Description :** Ce sont des sortes de cases cliquables contenant du texte et que nous allons insérer dans notre GTK_Menu_Bar (voir la figure suivante).



Une GTK_Menu_Bar contenant un GTK_Menu_Item

[Retour sur notre exemple](#)

Pour obtenir la barre ci-dessus, vous allez devoir créer un `Gtk_Menu_Item` et l'ajouter à votre barre à l'aide de l'une des trois méthodes `Append()`, `Prepend()` et `Insert()`. Voici la spécification du constructeur :

Code : Ada

```
procedure Gtk_New(Menu_Item : out Gtk_Menu_Item; Label : UTF8_String
                  := "");
```

Et voici ce que donnerait notre code :

Code : Ada

```
WITH Gtk.Main; USE Gtk.Main;
WITH Gtk.Window; USE Gtk.Window;
WITH Gtk.Menu_Bar; USE Gtk.Menu_Bar;
WITH Gtk.Menu_Item; USE Gtk.Menu_Item;
WITH Gtk.Enums; USE Gtk.Enums;

PROCEDURE MesMenus IS
    Win : Gtk.Window;
    Barre : Gtk.Menu_Bar;
    Item_Fichier : Gtk.Menu_Item;
BEGIN
    Init;
    --Création de la fenêtre
    Gtk_New(Win);
    Win.Set_Default_Size(250,25);
    Win.Set_Title("Mes menus");
    --Création de la barre de menu
    Gtk_New(Barre);
    Win.Add(Barre);
    --Création de l'item Fichier
    Gtk_New(Item_Fichier, "Fichier");
    Barre.Append(Item_Fichier);
    --Finalisation
    Win.Show_All;
    Main;
END MesMenus;
```



Erreur ! Le menu Fichier n'est pas vide : il n'existe pas ! Tout ce que vous avez fait pour l'instant, c'est ajouter du texte à votre barre. Il vous reste une dernière étape : créer un menu déroulant. Mais avant cela, jetons tout de même un rapide coup d'œil aux méthodes disponibles :

Code : Ada

```
procedure Gtk_New_With_Mnemonic(Menu_Item : out Gtk_Menu_Item;
                                 Label : UTF8_String);

procedure Set_Label(Menu_Item : access Gtk_Menu_Item_Record;
                   Label : String);
function Get_Label (Menu_Item : access Gtk_Menu_Item_Record)
                   return String;

procedure Set_Use_Underline(Menu_Item : access Gtk_Menu_Item_Record;
                           Setting : Boolean);
function Get_Use_Underline (Menu_Item : access Gtk_Menu_Item_Record)
                           return Boolean;
```

Ces méthodes devraient vous dire quelque chose désormais puisque nous les avons déjà vues lors des chapitres sur les étiquettes, les boutons ou les extensions. Elles vous permettront de redéfinir le texte de votre `Gtk.Item` ou d'utiliser des caractères soulignés (pratique si vous voulez créer des raccourcis claviers). Mais comme votre priorité est d'ajouter un menu à votre item, voici les méthodes qu'il vous faut :

Code : Ada

```
procedure Set_Submenu(Menu_Item : access Gtk_Menu_Item_Record;
                      Submenu : access Gtk_Widget_Record'Class);
function Get_Submenu (Menu_Item : access Gtk_Menu_Item_Record)
                     return Gtk_Widget;
```

Créer et ajouter un menu déroulant

Pour finaliser notre barre de menu, nous avons besoin d'un ultime widget : `Gtk.Menu`.

Fiche d'identité

- Widget : `Gtk.Menu`
- Package : `Gtk.Menu`
- Descendance : `Gtk.Widget` >> `Gtk.Container` >> `Gtk.Menu_Shell`
- Description : Ce widget est un menu déroulant pouvant contenir plusieurs items (voir la figure suivante).



Terminons notre exemple

Nous allons achever notre barre de menu. Pour cela nous allons :

1. créer un `Gtk.Menu` ;
2. l'attacher au `Gtk.Menu_Item` que nous avons appelé `Item_Fichier` à l'aide de la méthode `Set_Submenu()` vue précédemment ;
3. lui ajouter deux items : `Ouvrir` et `Fermer`.

Code : Ada

```
WITH Gtk.Main; USE Gtk.Main;
WITH Gtk.Window; USE Gtk.Window;
WITH Gtk.Menu_Bar; USE Gtk.Menu_Bar;
WITH Gtk.Menu; USE Gtk.Menu;
WITH Gtk.Menu_Item; USE Gtk.Menu_Item;

PROCEDURE MesMenus IS
    Win : Gtk.Window;
    Barre : Gtk.Menu_Bar;
    Menu_Fichier : Gtk.Menu;
    Item_Fichier : Gtk.Menu_Item;
    Item_Ouvrir : Gtk.Menu_Item;
    Item_Fermer : Gtk.Menu_Item;
BEGIN
    Init;
    --Création de la fenêtre
    Gtk_New(Win);
    Win.Set_Default_Size(250,25);
    Win.Set_Title("Mes menus");
```

```
--Création de la barre de menu
Gtk_New(Barre) ;
Win.Add(Barre) ;

--Création de l'item Fichier
Gtk_New(Item_Fichier, "Fichier") ;
Barre.Append(Item_Fichier) ;

--Création du menu Fichier
Gtk_New(Menu_Fichier) ;
Item_Fichier.Set_submenu(Menu_Fichier) ;

--Création de l'item Ouvrir
Gtk_New(Item_Ouvrir, "Ouvrir") ;
Menu_Fichier.append(Item_Ouvrir) ;

--Création de l'item Fermer
Gtk_New(Item_Fermer, "Fermer") ;
Menu_Fichier.append(Item_Fermer) ;

--Finalisation
Win.Show_All ;
Main ;

END MesMenus ;
```

Désormais, vous n'avez plus qu'à connecter les GTK Item à vos callbacks. Pour cela, vous aurez besoin du signal suivant : `Signal_Activate ou "activate"`. Celui-ci est émis dès lors que vous cliquez sur un item.

Améliorer la barre de menu

Vous connaissez désormais les rudiments nécessaires pour créer une barre de menu. Comme je ne cesse de le répéter depuis le début de la partie V, n'hésitez pas à ouvrir et lire les fichiers ads. Vous y trouverez toutes les méthodes disponibles ainsi que des commentaires généralement exhaustifs. Dans cette partie, nous allons voir comment améliorer nos menus à partir de ce que nous savons déjà ou à l'aide de nouveaux widgets.

Créer un menu en image

GTK est livré avec toute une batterie d'images permettant d'égayer vos menus et boutons. Vous pouvez par exemple obtenir un menu comme celui-ci :



Si vous êtes tentés, rien de plus simple. Abandonnez les `GTK_Menu_Item` au profit du widget fils : `Gtk_Image_Menu_Item`. Vous aurez deux possibilités pour créer vos items en image :

- Le créer comme précédemment puis lui ajouter une image.
- Utiliser une image prédéfinie de GTK.

Méthode artisanale

Dans le premier cas, utilisez `GTK_New()` comme vous savez le faire puis créez une image avec un `GTK_Image`. Il vous suffit ensuite d'attribuer l'image à votre item à l'aide de la méthode `Set_Image()` :

Code : Ada

```
procedure Set_Image(Menu_Item : access Gtk_Image_Menu_Item_Record;
                    Image      : access Gtk_Widget_Record'Class);
function Get_Image (Menu_Item : access Gtk_Image_Menu_Item_Record)
                   return Gtk_Widget;
```

Méthode pour ne pas réinventer la roue

Mais la solution de facilité est de faire appel au stock d'images intégrées dans le thème visuel de GTK. C'est très simple, il suffit d'utiliser la méthode `Gtk_New_From_Stock()` à la place de `GTK_New()` :

Code : Ada

```
procedure Gtk_New_From_Stock(Widget : out Gtk_Image_Menu_Item;
                           Stock_Id : string);
```

`Stock_Id` indique le nom de l'image et du texte à afficher. Ces noms sont nommés et commencent généralement par `"gtk-`. Vous trouverez la liste des valeurs possibles pour `Stock_Id` dans le package `GtkStock`. Pour notre menu Ouvrir, nous utiliserons `"gtk-open"` et pour Fermer, ce sera `"gtk-close"`.

C'est bien sympa tout ça mais je me retrouve avec des menus en Anglais?! 😐

Souvenez-vous que les `GTK_Image_Menu_Item` dérivent des `GTK_Menu_Item` et qu'ils héritent donc de leurs méthodes. Vous pourrez donc les renommer avec :

Code : Ada

```
procedure Set_Label(Menu_Item : access Gtk_Menu_Item_Record;
                    Label     : String);
```

Créer un sous-menu

Je voulais créer un sous-menu pour la sauvegarde car je prévois deux formats différents pour enregistrer mes documents. Comment faire ?

Vous n'avez besoin de rien de plus que ce que vous connaissez déjà. Tout d'abord, ajoutons un nouvel item pour la sauvegarde :

Code : Ada

```
... WITH Gtk.Image_Menu_Item ; USE Gtk.Image_Menu_Item ;

PROCEDURE MesMenus IS
  ... Item_Sauver : Gtk_Image_Menu_Item ;
BEGIN
  ...
  --Création de l'item Sauver
  Gtk_New_From_Stock(Item_Sauver, "gtk-save-as") ;
  Item_Sauver.set_label("Enregistrer au format") ;
  Menu_Fichier.Append(Item_Sauver) ;
  ...
END MesMenus ;
```

Puis, nous allons créer un second `Gtk_Menu` que nous ajouterons à l'item que nous venons de faire. Il ne restera plus qu'à ajouter deux nouveaux items :

Code : Ada

```

WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Menu_Bar ; USE Gtk.Menu_Bar ;
WITH Gtk.Menu ; USE Gtk.Menu ;
WITH Gtk.Menu_Item ; USE Gtk.Menu_Item ;
WITH Gtk.Image_Menu_Item ; USE Gtk.Image_Menu_Item ;

PROCEDURE MesMenus IS
  Win : GTK_Window ;
  Barre : GTK_Menu_Bar ;
  Menu_Fichier : Gtk_Menu ;
  Menu_Sauver : Gtk_Menu ;
  Item_Fichier : Gtk_Menu_Item ;
  Item_Ouvrir : Gtk_Image_Menu_Item ;
  Item_Fermer : Gtk_Image_Menu_Item ;
  Item_Sauver : Gtk_Image_Menu_Item ;
  Item_Format1 : Gtk_Menu_Item ;
  Item_Format2 : Gtk_Menu_Item ;
BEGIN
  Init ;
  --Création de la fenêtre
  Gtk_New(Win) ;
  Win.Set_Default_Size(250,25) ;
  Win.Set_Title("Mes menus") ;

  --Création de la barre de menu
 Gtk_New(Barre) ;
  Win.Add(Barre) ;

  --Création de l'item Fichier
  Gtk_New(Item_Fichier,"Fichier") ;
  Barre.Append(Item_Fichier) ;

  --Création du menu Fichier
  Gtk_New(Menu_Fichier) ;
  Item_Fichier.Set_Submenu(Menu_Fichier) ;

  --Création de l'item Ouvrir
  Gtk_New_From_Stock(Item_Ouvrir,"gtk-open") ;
  Item_Ouvrir.set_label("Ouvrir") ;
  Menu_Fichier.append(Item_Ouvrir) ;

  --Création de l'item Fermer
  Gtk_New_From_Stock(Item_Fermer,"gtk-close") ;
  Item_Fermer.set_label("Fermer") ;
  Menu_Fichier.append(Item_Fermer) ;

  --Création de l'item Sauver
  Gtk_New_From_Stock(Item_Sauver,"gtk-save-as") ;
  Item_Sauver.set_label("Enregistrer au format") ;
  Menu_Fichier.append(Item_Sauver) ;

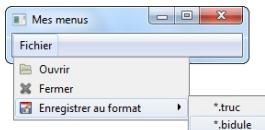
  --Création du sous-menu
  Gtk_New(Menu_Sauver) ;
  Item_Sauver.Set_Submenu(Menu_Sauver) ;

  --Création de l'item de format n°1
  Gtk_New(Item_Format1,"*.truc") ;
  Menu_Sauver.append(Item_Format1) ;

  --Création de l'item de format n°2
  Gtk_New(Item_Format2,"*.bidule") ;
  Menu_Sauver.append(Item_Format2) ;

  --Finalisation
  Win.Show_All ;
  Main ;
END MesMenus ;

```



Proposer un item à cocher dans le menu



Je souhaite proposer une option "Afficher le score" dans mon menu que le joueur pourra cocher ou décocher. Comment faire ?

Encore une fois, rien de plus simple. Nous allons faire appel au widget `Gtk_Check_Menu_Item` au lieu de `Gtk_Menu_Item`. Il s'agit d'un type dérivé bien entendu ce qui nous permettra de réutiliser les méthodes du père. Voici ce que cela pourrait donner :

Code : Ada

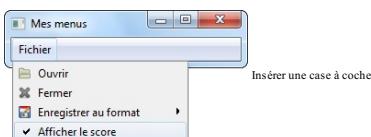
```

...
WITH Gtk.Check_Menu_Item ; USE Gtk.Check_Menu_Item ;

PROCEDURE MesMenus IS
  ...
  Item_Afficher : Gtk_Check_Menu_Item ;
BEGIN
  ...
  --Création de l'item Afficher score
  Gtk_New(Item_Afficher,"Afficher le score") ;
  Menu_Fichier.append(Item_Afficher) ;
  ...
END MesMenus ;

```

Admirez le dernier item du menu : une case à cocher ! (Voir la figure suivante.)



Dans le package `Gtk.Check_Menu_Item`, vous trouverez la plupart des méthodes additionnelles, mais voici celles qui vous seront essentielles :

Code : Ada

```

procedure Set_Active(Check_Menu_Item :          access
                      Gtk_Check_Menu_Item_Record;
                      Is_Active : Boolean);
function Get_Active (Check_Menu_Item :          access
                      Gtk_Check_Menu_Item_Record) return Boolean;

```

Avec `Set_Active()`, vous pourrez cocher ou décocher votre item, tandis qu'avec `Get_Active()`, vous pourrez connaître son état. Enfin, chaque fois que l'utilisateur coche ou décoche votre item, le signal `Signal_Toggled` ou "`toggled`" est émis.

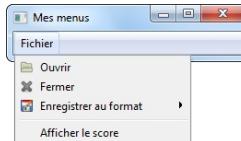


Vous aurez compris que les items à cocher (`Gtk_Check_Menu_Item`) sont très similaires aux boutons à cocher

 (Gtk_Check_Button) vus au chapitre 5. De la même manière, il existe des items radio, semblables aux boutons radio (Gtk_Radio_Button). Ils s'appellent tout simplement Gtk_Radio_Menu_Item.

Organiser vos menus

Lorsque vos menus commenceront à devenir volumineux, il sera judicieux d'utiliser des séparateurs pour que l'utilisateur visualise rapidement les groupes logiques d'item. Aussi aurez-vous besoin du widget Gtk_Separator_Menu_Item. Vous constaterez, sur la figure ci-dessous, la présence d'un petit trait séparant l'item "Afficher le score" des trois autres items.



La barre d'icônes

Fiches d'identité

Pour réaliser une barre d'icônes, nous aurons besoin des deux widgets GTK_Toolbar et GTK_Tool_Button. On retrouve ainsi une architecture semblable aux barres de menu : une barre servant de conteneur et des objets à y insérer.

Fiche d'identité des GTK_Toolbar

- **Widget :** GTK_Toolbar
- **Package :** GTK.Toolbar
- **Descendance :** GTK_Widget >> GTK.Container
- **Description :** La GTK_Toolbar est la barre dans laquelle nous placerons nos icônes.

Fiche d'identité des GTK_Tool_Button

- **Widget :** GTK_Tool_Button
- **Package :** GTK.Tool_Button
- **Descendance :** GTK_Widget >> GTK.Container >> GTK_bin >> GTK_Tool_Item
- **Description :** Les GTK_Tool_Button sont les boutons des icônes à insérer.

Créer notre barre d'icônes

Les méthodes élémentaires

Comme toujours, vous devez commencer par construire la barre à l'aide de la méthode :

Code : Ada

```
procedure Gtk_New (Widget : out Gtk_Toolbar);
```

Puis vous devrez construire votre icône avec les méthodes GTK_New() ou GTK_New_From_Stock() :

Code : Ada

```
procedure Gtk_New(Button : out Gtk_Tool_Button;
                  Icon_Widget : Gtk.Widget.Gtk_Widget := null;
                  Label : String := "");
procedure Gtk_New_From_Stock(Button : out Gtk_Tool_Button;
                             Stock_Id : String);
```

En revanche, pour ajouter un icône à la barre, vous ne disposerez plus de méthode Append() ou Prepend() (celles-ci étant obsolètes). Vous devrez donc utiliser la méthode :

Code : Ada

```
procedure Insert(Toolbar : access Gtk_Toolbar_Record;
                 Item : access Gtk_Tool_Item_Record'Class;
                 Pos : Gint := -1);
```

Si le paramètre Pos vaut 0, votre icône sera placé en première position, s'il vaut 1 en deuxième position... Et si vous n'avez aucune idée de l'emplacement où cet icône doit être située, donnez-lui comme position : GTK le placera en bout de liste.

Exemple

Pour mieux comprendre, rien de tel qu'un exemple. Nous allons construire un programme comportant une barre d'icône et deux icônes. Le code ci-dessous devrait être suffisamment simple et commenté pour que vous le déchiffriez seuls :

Code : Ada

```
WITH Gtk.Main ; USE Gtk.Main ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Toolbar ; USE Gtk.Toolbar ;
WITH Gtk.Tool_button ; USE Gtk.Tool_button ;

PROCEDURE MesMenus IS
  Win : Gtk.Window ;
  Barre : Gtk.Toolbar ;
  Icon_Ouvrir : Gtk_Tool_Button ;
  Icon_Fermer : Gtk_Tool_Button ;
BEGIN
  Init ;
  --Création de la fenêtre
  Gtk_New(Win) ;
  Win.Set_Default_Size(250,25) ;
  Win.Set_Title("Mes menus") ;

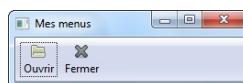
  --Création barre d'outils
  Gtk_New(Barre) ;
  Win.add(barre) ;

  --Création de l'icône ouvrir
  Gtk_New_From_Stock(Icon_Ouvrir,"gtk-open") ;
  icon_ouvrir.set_label("Ouvrir") ;
  Barre.Insert(Icon_Ouvrir) ;

  --Création de l'icône fermer
  Gtk_New_From_Stock(Icon_Fermer,"gtk-close") ;
  icon_fermer.set_label("Fermer") ;
  Barre.Insert(Icon_fermer) ;

  --Finalisation
  Win.Show_All ;
  Main ;
END MesMenus ;
```

Voici, sur la figure suivante, le résultat en image.



Une barre d'icônes

Pour plus de clarté, les icônes ne sont reliés à aucun callback, mais il est évident que vous devrez connecter chacun d'entre eux à sa méthode. Le signal émis par les `Gtk_Tool_Button` lorsque l'utilisateur clique dessus se nomme `Signal_Clicked` ou `"clicked"`.

Améliorer les barres d'icônes

D'autres icônes



Existe-t-il d'autres types d'icônes, à la manière des items de menu ?

Bien sûr. Il n'existe pas que les `Gtk_Tool_Button`, sinon les créateurs de GTK n'auraient pas créé de classe mère `Gtk_Tool_Item`. Il existe :

- des icônes séparateurs pour séparer des groupes d'icônes : les `Gtk_Separator_Tool_Item`;
- des icônes à bascule pour activer ou désactiver un option : les `Gtk_Toggle_Tool_Button`;
- des icônes radio pour choisir entre un groupe d'options : les `Gtk_Radio_Tool_Button`.

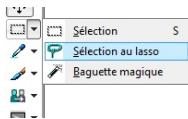
Enfin, il existe un dernier type d'icône. Il s'agit d'icônes ouvrant des menus : les `Gtk_Menu_Tool_Button`. Pour leur adjoindre un menu, il faudra utiliser la méthode suivante :

Code : Ada

```
procedure Set_Menu(Button : access Gtk_Menu_Tool_Button_Record;
                   Menu   : access Gtk_Menu_Record'Class);
```

Ce genre d'icône vous sera très utile si vous comptez créer de nombreux outils pour votre logiciel. Ainsi, pour les logiciels de dessin, les outils de sélection par zone, à main levée et par couleur sont généralement rassemblés sous un même icône de sélection.

Sur la figure suivante, un exemple d'utilisation d'un icône-menu avec le logiciel Paint Shop Pro



Icône-menu sur Paint Shop Pro

Ajouter une info-bulle

Si vous explorez le package `Gtk.Tool_Button`, vous serez déçus de ne trouver que très peu de méthodes, hormis les éléments (`Get/Set_Label()`, `(Get/Set)_Icon_Widget()`, `(Get/Set)_Stock_Id()`...) permettant d'obtenir ou d'édition les paramètres essentiels du bouton. Mais en fouillant dans le package père `Gtk.Tool_Item`, vous trouverez les méthodes suivantes :

Code : Ada

```
procedure Set_Tooltip_Text  (Tool_Item :          access
                           Gtk_Tool_Item_Record;
                           Text      : UTF8_String);
procedure Set_Tooltip_Markup(Tool_Item :          access
                           Gtk_Tool_Item_Record;
                           Markup   : UTF8_String);
```

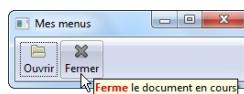
Celles-ci permettent de créer des info-bulles pour vos icônes expliquant leur action. La seconde méthode permet même d'utiliser des balises de mise en forme (voir le chapitre sur les étiquettes pour rappel). Et vous n'avez pas besoin de faire appel à ce package puisque par dérivation, les `Gtk_Tool_Button` héritent de ces méthodes. Exemple :

Code : Ada

```
WITH Gtk.Main ;           USE Gtk.Main ;
WITH Gtk.Window ;         USE Gtk.Window ;
WITH Gtk.Toolbar ;        USE Gtk.Toolbar ;
WITH Gtk.Tool_Button ;    USE Gtk.Tool_Button ;

PROCEDURE MesIcônes IS
  Win          : GTK_Window ;
  Barre        : Gtk_Toolbar ;
  Icon_Ouvrir  : Gtk_Tool_Button ;
  Icon_Fermer   : Gtk_Tool_Button ;
BEGIN
  Init ;
  --Création de la fenêtre
  Gtk_New(Win) ;
  Win.Set_Default_Size(250,25) ;
  Win.Set_Title("Mes menus") ;
  --Création barre d'outils
  Gtk_New(Barre) ;
  Win.add(Barre) ;
  --Création de l'icône Ouvrir
  Gtk_New_From_Stock(Icon_Ouvrir,"gtk-open") ;
  Icon_Ouvrir.Set_Label("Ouvrir") ;
  Icon_Ouvrir.Set_Tooltip_Text("Permet d'ouvrir un document") ;
  Barre.Insert(Icon_Ouvrir) ;
  --Création de l'icône Fermer
  Gtk_New_From_Stock(Icon_Fermer,"gtk-close") ;
  Icon_Fermer.Set_Label("Fermer") ;
  Icon_Fermer.Set_Tooltip_Markup("<span>Ferme</span> le document en cours") ;
  Barre.Insert(Icon_Fermer) ;
  --Finalisation
  Win.Show_All ;
  Main ;
END MesIcônes ;
```

Sur la figure suivante, on constate que lorsque l'icône est survolé par la souris, une info-bulle apparaît.



Une info-bulle avec mise en forme

Améliorez la barre

De nombreuses méthodes liées aux `Gtk.Toolbar` sont désormais obsolètes, si bien qu'il n'en reste que peu. Mais voici quelques méthodes qui pourraient vous intéresser :

Code : Ada

```
procedure Set_Orientation(Toolbar : access Ctk_Toolbar_Record;
                         Orientation : Gtk_Orientation);
function Get_Orientation(Toolbar : access Gtk_Toolbar_Record) return
```

```

    Gtk_Orientation;
procedure Set_Tooltips(Toolbar : access Gtk_Toolbar_Record;
                      Enable   : Boolean);
function Get_Tooltips(Toolbar : access Gtk_Toolbar_Record) return
Boolean;
procedure Set_Show_Arrow(Toolbar  : access Gtk_Toolbar_Record;
                        Show Arrow : Boolean := True);
function Get_Show_Arrow(Toolbar : access Gtk_Toolbar_Record)
return Boolean;

```

Avec `Set_Orientation()` vous pourrez indiquer si vos icônes sont alignés horizontalement (`Orientation` vaut alors `Orientation_Horizontal`) ou verticalement (`Orientation_Vertical`). Avec `Set_Tooltips()` vous pourrez autoriser ou non l'affichage des info-bulles ; enfin avec `Set_Show_Arrow()`, vous pourrez afficher ou non une flèche en bout de barre. Si votre barre contient trop d'icônes, cette flèche vous permettra d'ouvrir un petit menu pour avoir accès aux icônes cachés. Cette flèche n'apparaît qu'en cas de besoin et est activée par défaut.

 Les barres d'icônes sont aussi appelées barres d'outils. Beaucoup de logiciels permettent de déplacer ces barres d'outil pour les placer en haut, à gauche ou en bas de la fenêtre ou de les en détacher. Pour obtenir cet effet, vous pouvez combiner votre `Gtk_Toolbar` avec une `Gtk_Handle_Box` (voir chapitre sur les boîtes détachables).

Combiner menus et icônes

Dans certains logiciels, comme ceux de dessin, les icônes sont essentielles et donnent accès à des outils inatteignables autrement, comme le pinceau ou les outils de sélection. Mais la plupart du temps, ces icônes sont des raccourcis de menus déjà existants. Si nous essayons de combiner la barre de menus et la barre d'icônes que nous venons de créer, nous allons voir apparaître de sacrés doublons ! Il faudra créer un `Gtk_Menu_Item` et un `Gtk_ToolButton` pour ouvrir un document, pour le fermer, pour en créer un nouveau, pour le sauvegarder ... sans parler des callbacks qu'il faudra connecter deux fois. Une solution à ce problème est de faire appel aux actions, aussi appelées `Gtk_Action`.

Fiche d'identité

- **Widget :** `Gtk_Action`
- **Package :** `Gtk.Action`
- **Descendance :** il ne s'agit pas d'un widget. Toutefois il dérive des `Gtk.Object`, tout comme les `Gtk_Widget`.
- **Description :** définit un objet de type action qui pourra être réutilisé par la suite pour créer un item de menu ou un icône.

Mode d'emploi

L'emploi des actions pourrait être déconcertant, mais vous avez désormais l'expérience suffisante pour les aborder en toute sérénité. La partie la plus complexe est, une fois n'est pas coutume, le constructeur. Celui-ci va cumuler tous les paramètres nécessaires aux icônes et items de menu :

Code : Ada

```

procedure Gtk_New(Action  : out Gtk_Action;
                  Name   : String;
                  Label  : String;
                  Tooltip : String := "";
                  Stock_Id : String := "");

```

Le paramètre `Label` est le texte qui sera affiché dans le menu ou sous l'icône ; `Tooltip` correspond au texte qui sera affiché dans l'info-bulle ; `Stock_Id` est bien entendu le nom de l'image (`"gtk-open"`, `"gtk-close"`...). Le paramètre `Name` correspond à un nom que vous devez donner à cette action. Ce nom doit être unique, de même que chaque signal à un nom unique.

La connexion aux callbacks se fait de la même façon que pour les widgets, toutefois faites attention à linstanciation de vos packages : les actions ne sont pas des widgets ! Il s'agit d'un type d'objet frère mais distinct ; widgets et actions sont tous deux des `Gtk_Object`.

Une fois vos actions créées et connectées, leur transformation en menus, en items de menu ou en icônes pourra se faire avec l'une des méthodes suivantes :

Code : Ada

```

function Create_Menu      (Action : access Gtk_Action_Record) return
Gtk_Widget;
function Create_Menu_Item (Action : access Gtk_Action_Record) return
Gtk_Widget;
function Create_Tool_Item (Action : access Gtk_Action_Record) return
Gtk_Widget;

```

Toutefois, ces fonctions de conversion transformeront votre action en widget, mais pas spécifiquement en item, menu ou icône. Pour mieux comprendre, reprenons nos codes précédents et réalisons une fenêtre contenant une barre de menu et une barre d'icônes. Nous nous limiterons aux deux actions Ouvrir et Nouveau :

Code : Ada

```

WITH Gtk.Main ;
WITH Gtk.Window ;
WITH Gtk.Box ;
WITH Gtk.Menu_Bar ;
WITH Gtk.Menu ;
WITH Gtk.Menu_Item ;
WITH Gtk.Toolbar ;
WITH Gtk.Tool_button ;
WITH Glib.Convert ;
WITH Gtk.Action ;
USE Gtk.Main ;
USE Gtk.Window ;
USE Gtk.Box ;
USE Gtk.Menu_Bar ;
USE Gtk.Menu ;
USE Gtk.Menu_Item ;
USE Gtk.Toolbar ;
USE Gtk.Tool_button ;
USE Glib.Convert ;
USE Gtk.Action ;

```



```

PROCEDURE DoubleMenu IS
  Win      : GTK_Window ;
  Vbox     : Gtk_Vbox ;
  Barre1   : GTK_Menu_Bar ;
  Barre2   : GTK_Toolbar ;
  Menu_Fichier : Gtk_Menu ;
  Item_Fichier : Gtk_Menu_Item ;
  Action_Nouveau : Gtk_Action ;
  Action_Ouvrir : Gtk_Action ;
BEGIN
  Init ;
  --Création de la fenêtre
  Grk_New(Win) ;
  Win.Set_Default_Size(250,25) ;
  Win.Set_Title("Mes menus") ;

  --Création boîte
  Gtk_New_Vbox(Vbox) ;
  Win.Add(Vbox) ;

  --Création de la barre de menu
  Grk_New(Barre1) ;
  Vbox.Pack_Start(Barre1) ;

  --Création barre d'icônes
  Grk_New(Barre2) ;
  Vbox.Pack_Start(Barre2) ;

  --Création de l'item Fichier
  Grk_New(Item_Fichier,"Fichier") ;
  Barre1.Append(Item_Fichier) ;

  --Création du menu Fichier
  Grk_New(Menu_Fichier) ;
  Item_Fichier.Set_Submenu(Menu_Fichier) ;

  --Création de l'action ouvrir
  Grk_new(Action_Ouvrir,"action-open","Ouvrir","Ouvre un document
existent","gtk-open") ;
  Menu_Fichier.Append(GTK_Menu_item(Action_Ouvrir.Create_Menu_Item)) ;

```

```
Barre2.insert(gtk_tool_button(Action_Ouvrir.Create_Tool_Item)) ;
    --Création de l'action nouveau
Gtk_new(Action_nouveau,"action-new","Nouveau",locale_to_utf8("Crée
un nouveau document"),"gtk-new") ;
Menu_Fichier.Append(GTK_Menu_item(Action_Nouveau.Create_Menu_Item))
; Barre2.insert(gtk_tool_button(Action_nouveau.Create_Tool_Item)) ;

--Finalisation
Win.Show_All ;
Main ;

```

Comme vous pouvez le constater, pour créer deux items de menu et deux icônes (soit quatre widgets), je n'ai eu besoin que de deux actions (et il n'aurait fallu que deux connexions de callback au lieu de quatre, non écrits par soucis de clarté). J'attire toutefois votre attention sur les lignes 52,53,57 et 58 :

Code : Ada

```
Menu_Fichier.Append(GTK_Menu_item(Action_Ouvrir.Create_Menu_Item)) ;
Barre2.insert(gtk_tool_button(Action_Ouvrir.Create_Tool_Item)) ;
```

Code : Ada

```
Menu_Fichier.Append(GTK_Menu_item(Action_Nouveau.Create_Menu_Item))
; Barre2.insert(gtk_tool_button(Action_nouveau.Create_Tool_Item)) ;
```

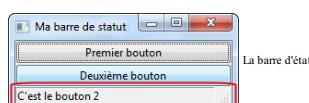
Aux lignes 52 et 57, j'ajoute au menu Fichier mes deux actions que je convertis en widget grâce à `Create_Menu_Item()`. Toutefois, la méthode `Append()` n'attend pas un `Gtk.Widget` mais un `Gtk.Menu.Item` ! C'est pourquoi il faut effectuer une conversion supplémentaire. Le phénomène est le même aux lignes 53 et 58 où j'ai convertis les actions en widget grâce à `Create_Tool_Item()` puis en icône avec `gtk_tool_button()`.

La barre de statut

Fiche d'identité

- **Widget** : `GTK_Status_Bar`
- **Package** : `GTK_Status_Bar`
- **Désendance** : `GTK_Widget` >> `GTK.Container` >> `GTK_Box` >> `GTK_Hbox`
- **Description** : La barre de statut (ou barre d'état) affiche des informations en continu pour l'utilisateur. Elle est généralement située au bas de la fenêtre.

Sur la figure suivante, au survol du second bouton, la barre de statut affiche «C'est le bouton 2»



Méthodes et fonctionnement

Méthodes pour l'apparence

Les barres de statut disposent de très peu de méthodes propres. Voici celles concernant l'apparence :

Code : Ada

```
function Get_Has_Resize_Grip(Self : access Gtk_Status_Bar_Record) return Boolean;
procedure Set_Has_Resize_Grip(Self : access Gtk_Status_Bar_Record;
                             Setting : Boolean);
function Get_Orientation(Self : access Gtk_Status_Bar_Record) return Gtk.Enums.Gtk_Orientation;
procedure Set_Orientation(Self : access Gtk_Status_Bar_Record;
                         Orientation : Gtk.Enums.Gtk_Orientation);
```

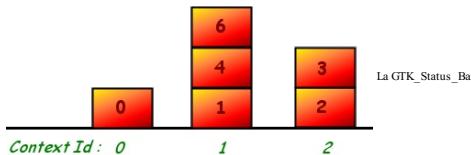
La méthode `Set_Resize_Grip()` vous permet d'indiquer si la barre contient ou non un petit carré à droite pour redimensionner la fenêtre. Par défaut, ce carré est affiché. Je ne m'attarde plus sur les méthodes `Get_Orientation()` et `Set_Orientation()` que nous avons vues et revues (retournez à la partie précédente sur la barre d'icônes si vous avez déjà oublié).

Fonctionnement

Mais le but d'une barre d'état est avant tout d'afficher des messages. Alors vous devez d'abord comprendre comment sont gérés ces messages. À chaque fois que vous souhaitez afficher un message, vous devrez l'ajouter à la liste de messages de la `GTK_Status_Bar`. Celle-ci lui attribuera un numéro d'identification de message, qui n'est autre qu'un entier de type `Message_Id`. Mais ce n'est pas tout ! Vous devrez également fournir, en plus de la chaîne de caractères, un numéro d'identification de contexte qui est également un entier, mais de type `Context_Id`. Ce nouveau numéro d'identification permet de classer les messages selon leur «contexte» : les messages d'erreurs porteront par exemple le numéro 0 et les messages d'aide le numéro 1.

Cet identifiant nous sera inutile car nous ne pousserons pas la difficulté si loin mais nous devons tout de même le fournir et nous en souvenir car il nous sera demandé lorsque nous souhaiterons retirer notre dernier message. En effet, la liste de messages fonctionne à la manière d'une liste de piles (voir [chapitre sur les TAD](#)) : pour retirer un message, vous fournissez le `Context_Id` et GTK retire de la pile correspondante le dernier message posté.

Sur la figure suivante, la représentation de la liste des messages dans une `GTK_Status_Bar`.



Sur le schéma précédent, on peut deviner l'ordre d'arrivée des messages et leur classement :

1. Ajout du message n°0 dans le Context_Id n°0
2. Ajout du message n°1 dans le Context_Id n°1
3. Ajout du message n°2 dans le Context_Id n°2
4. Ajout du message n°3 dans le Context_Id n°2
5. Ajout du message n°4 dans le Context_Id n°1
6. Ajout du message n°5 dans un Context_Id inconnu
7. Suppression du message n°5
8. Ajout du message n°6 dans le Context_Id n°1

Le message affiché en ce moment est donc le n°6, c'est le plus vieux. S'il était supprimé, ce serait le 4, toujours dans le `Context_Id` n°1, puis le 3 et ainsi de suite.

Méthodes essentielles

Venons-en maintenant aux méthodes :

Code : Ada

```
function Push(Statusbar : access Gtk_Status_Bar_Record;
             Context   : Context_Id;
             Text      : UTF8_String) return Message_Id;
procedure Pop(Statusbar : access Gtk_Status_Bar_Record;
             Context   : Context_Id);
procedure Remove(Statusbar : access Gtk_Status_Bar_Record;
                Context   : Context_Id;
                Message   : Message_Id);
procedure Remove_All(Statusbar : access Gtk_Status_Bar_Record;
                     Context   : Context_Id);
```

Si vous vous souvenez du chapitre sur les piles, les noms de ces procédures devraient vous sembler familiers : Push () ajoute un message à une pile ; Pop () retire l'élément au sommet d'une pile ; Remove () permet de supprimer un message précis d'une pile précise ; Remove_all () vide toute une pile. Vous remarquerez que toutes ces méthodes exigent un Context_Id, c'est-à-dire le numéro de la pile désirée. De plus, faites attention car Push () n'est pas une procédure mais une fonction qui renvoie le numéro du message créé nécessaire si vous souhaitez utiliser Remove () .

Exemples d'utilisation

Exemple basique

Contrairement aux autres widgets, je vais devoir vous présenter les barres de statut avec quelques callbacks, sans quoi cela n'aurait aucun intérêt. Voici donc le code écrit pour réaliser la fenêtre d'exemple contenant deux boutons et une barre d'état :

Code : Ada

```
WITH Gtk.Main; USE Gtk.Main;
WITH Gtk.Window; USE Gtk.Window;
WITH Gtk.Box; USE Gtk.Box;
WITH Gtk.Button; USE Gtk.Button;
WITH Glib.Convert; USE Glib.Convert;
WITH Gtk.Status_Bar; USE Gtk.Status_Bar;
WITH Gtk.Handlers; USE Gtk.Handlers;

PROCEDURE MaBarreDeStatut IS
    --Objets
    Win : Gtk.Window;
    Vbox : Gtk_Vbox;
    Barre : Gtk_Status_Bar;
    Btn1, Btn2 : Gtk_Button;
    Msg0 : Message_Id;

    --Type nécessaire aux callbacks
    TYPE T_Data IS RECORD
        Barre : Gtk_Status_Bar;
        Id : Integer;
    END RECORD;

    --Packages
    PACKAGE P_Callback IS
        NEW
        Gtk.Handlers.User_Callback(Gtk_Button_Record,T_Data);
        USE P_Callback;
    END P_Callback;

    --Callbacks
    PROCEDURE Active_Barre(Emetteur : ACCESS Gtk_Button_Record'Class
                           Data : T_Data) IS
        PRAGMA Unreferenced(Emetteur);
        Msg1 : Message_Id;
    BEGIN
        CASE Data.Id IS
            WHEN 1 => Msg1 := Data.Barre.Push(l,"C'est le bouton 1");
            WHEN 2 => Msg1 := Data.Barre.Push(l,"C'est le bouton 2");
            WHEN OTHERS => NULL;
        END CASE;
    END Active_Barre;

    PROCEDURE Desactive_Barre(Emetteur : ACCESS Gtk_Button_Record'Class
                           Data : T_Data) IS
        PRAGMA Unreferenced(Emetteur);
    BEGIN
        Data.Barre.Pop(1);
        END Desactive_Barre;

    BEGIN
        Init;
        --Création de la fenêtre
        Gtk_New(Win);
        Win.Set_Default_Size(250,300);
        Win.Set_Title("Ma barre de statut");

        --Création de la boîte
        Gtk_New_Vbox(Vbox);
        Win.Add(Vbox);

        --Création de la barre de statut
       Gtk_New(Barre);
        Msg0 := Barre.Push(0,"Passer le curseur sur un bouton");
        Vbox.pack_end(Barre, expand => false);

        --Création des boutons
        Gtk_New(Btn1,"Premier bouton");
        Connect(Btn1,"enter", Active_Barre'ACCESS, (Barre,1));
        Connect(Btn1,"leave", Desactive_Barre'ACCESS, (Barre,1));
        Vbox.Pack_Start(Btn1);

        Gtk_New(Btn2,Locale.To_UTF8("Deuxième bouton"));
        Connect(Btn2,"enter", Active_Barre'ACCESS, (Barre,2));
        Connect(Btn2,"leave", Desactive_Barre'ACCESS, (Barre,2));
        Vbox.Pack_Start(Btn2);

        --Finalisation
        Win.Show_All;
        Main;
    END MaBarreDeStatut;
```

Pour chaque bouton, il effectue deux connexions. Pour le signal "enter" (émis lorsque la souris entre dans le bouton) je déclenche le callback Active_Barre () qui affiche un message particulier : pour le signal "leave" (émis lorsque la souris quitte le bouton) je déclenche le callback Desactive_Barre () qui efface le message écrit. Le premier callback utilise la méthode push (), le second utilise pop (). Ainsi, les messages concernant les boutons n'apparaissent que si la souris est au-dessus et disparaissent sinon.

Une petite amélioration



Une barre de statut, c'est bien gentil mais ça n'est pas fondamental : tant de travail pour un tout petit message en bas ! Est-ce si utile ?

Jadmet que c'est du signolage. Mais une barre d'état peut être plus utile que vous ne le pensez. Si vous avez été attentifs à la fiche d'identité, vous avez dû (ou vous aurez du) remarquer une bizarrerie : les GTK_Status_Bar dérivent des GTK_HBox ! Étonnant, non ? Où mais voilà qui va nous être utile. Notre barre de statut peut donc contenir d'autres widgets ! Alors pourquoi ne pas y ajouter une barre de progression pour un éventuel téléchargement ainsi qu'une case à cocher pour une option ? (Voir la figure suivante.)

Code : Ada

```
WITH Gtk.Main; USE Gtk.Main;
WITH Gtk.Window; USE Gtk.Window;
WITH Gtk.Box; USE Gtk.Box;
WITH Gtk.Button; USE Gtk.Button;
WITH Glib.Convert; USE Glib.Convert;
WITH Gtk.Status_Bar; USE Gtk.Status_Bar;
WITH Gtk.ProgressBar; USE Gtk.ProgressBar;
--fin du code à lire
```

```

WITH Gtk.Check_Button ;           USE Gtk.Check_Button ;
WITH Gtk.Handlers ;
```

PROCEDURE MaBarreDeStatus IS

--Objets

Win : GTK.Window ;
Vbox : Gtk_Vbox ;
Barre : GTK_Status_Bar ;
Bt1, Bt2 : Gtk_Button ;
Msg0 : Message_Id ;
Progress : Gtk_Progress_Bar ;
Option : Gtk_Check_Button ;

--Type nécessaire aux callbacks

TYPE T_Data IS RECORD
 Barre : GTK_Status_Bar ;
 Id : Integer ;
END RECORD ;

--Packages

PACKAGE P_Callback IS

NEW
 Gtk.Handlers.User_Callback(Gtk_Button_Record,T_Data) ;
 USE P_Callback ;

--Callbacks

PROCEDURE Active_Barre(Emetteur : ACCESS Gtk_Button_Record'Class)

; Data : T_Data) IS
 PRAGMA Unreferenced(Emetteur) ;
 Msg1 : Message_Id ;
BEGIN

BEGIN Data.Id IS
 WHEN 1 => Msg1 := Data.Barre.Push(1,"C'est le bouton 1") ;
 WHEN 2 => Msg1 := Data.Barre.Push(1,"C'est le bouton 2") ;
 WHEN OTHERS => NULL ;
END CASE ;
END Active_Barre ;

PROCEDURE Desactive_Barre(Emetteur : ACCESS Gtk_Button_Record'Class)
; Data : T_Data) IS
 PRAGMA Unreferenced(Emetteur) ;
BEGIN
 Data.Barre.Pop(1) ;
END Desactive_Barre ;

BEGIN

Init ;
--Création de la fenêtre
Gtk_New(Win) ;
Win.Set_Default_Size(400,300) ;
Win.Set_Title("Ma barre de statut") ;

--Création de la boîte
Gtk_New_Vbox(Vbox) ;
Win.Add(Vbox) ;

--Création de la barre de statut
Gtk_New(Barre) ;
Msg0 := Barre.Push(0,"Passez le curseur sur un bouton") ;
Vbox.pack_end(Barre, expand => false) ;

--Création des boutons
Gtk_New(Bt1,"Premier bouton") ;
Connect(Bt1,"enter", Active_Barre'ACCESS, (Barre,1)) ;
Connect(Bt1,"leave", Desactive_Barre'ACCESS, (Barre,1)) ;
Vbox.Pack_Start(Bt1) ;

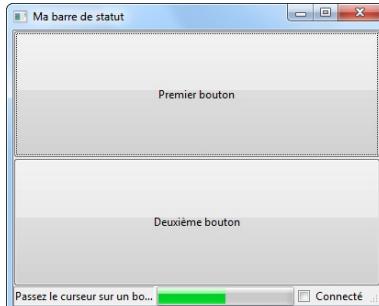
Gtk_New(Bt2,Locale.To_Utf8("Deuxième bouton")) ;
Connect(Bt2,"enter", Active_Barre'ACCESS, (Barre,2)) ;
Connect(Bt2,"leave", Desactive_Barre'ACCESS, (Barre,2)) ;
Vbox.Pack_Start(Bt2) ;

--Création d'une case à cocher
Gtk_New(Option,Locale.To_Utf8("Connecté")) ;
Barre.Pack_End(Option, Expand => False, Fill => False) ;

--Création d'une barre de progression
Gtk_New(Progress) ;
Progress.set_fraction(0.5) ;
Barre.Pack_End(Progress,Expand=> False, Fill => False) ;

--Finalisation
Win.Show_All ;
Main ;

END MaBarreDeStatus ;



La barre de statut avec une barre de progression

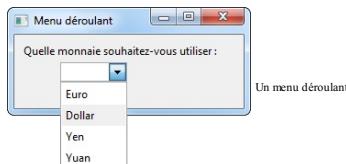
Maintenant, libre à vous de proposer des barres d'état avec davantage d'options. Pourquoi ne pas y intégrer l'heure, des statistiques concernant votre document, un outil de zoom ou des boutons pour remplacer les boîtes de dialogue !

Le menu déroulant

Fiche d'identité

- **Widget :** GTK_Combo_Box
- **Package :** GTK.Combo_Box
- **Descendance :** GTK_Widget >> GTK_Container >> GTK_Bin
- **Description :** Le GTK_Combo_Box se présente sous la forme d'un menu déroulant permettant un choix entre différents textes.

Sur la figure suivante, le menu déroulant permet de choisir entre différentes monnaies.



Méthodes

De nombreuses méthodes existent pour les `GTK_Combo_Box`, mais nous allons nous concentrer sur le cas courant : un menu déroulant ne proposant que des lignes de texte. Auquel cas, nous devrons utiliser le constructeur suivant :

Code : Ada

```
procedure Gtk_New_Text(Combo : out Gtk_Combo_Box);
```

L'ajout de texte se fait à l'aide de méthodes de type Append-Prepend-Insert comme vous devez vous en douter :

Code : Ada

```
procedure Append_Text(Combo_Box : access Gtk_Combo_Box_Record;
                      Text : String);
procedure Prepend_Text(Combo_Box : access Gtk_Combo_Box_Record;
                       Text : String);
procedure Insert_Text(Combo_Box : access Gtk_Combo_Box_Record;
                      Position : Gint;
                      Text : String);
```

Pour la suppression d'une ligne de texte, vous devrez faire appel à la méthode `Remove_Text()` et renseigner la position de la ligne souhaitée :

Code : Ada

```
procedure Remove_Text(Combo_Box : access Gtk_Combo_Box_Record;
                      Position : Gint);
```

Enfin, vous aurez besoin pour vos callbacks de connaître la valeur sélectionnée par l'utilisateur. Deux méthodes existent : `Get_Active()` renverra le numéro de la valeur choisie (0 pour la première valeur, 1 pour la seconde... -1 si aucune n'est sélectionnée) ; `Get_Active_Text()` renverra quant à elle le texte sélectionné (si aucune valeur n'a été sélectionnée, elle renverra un `String` vide). Inversement, vous pourrez définir le texte activé avec `Set_Active()`, par exemple pour définir une valeur par défaut. Attention toutefois, cette méthode émet automatiquement le signal `Signal_Changed / "changed"` indiquant que la valeur a été modifiée.

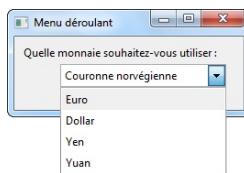
Code : Ada

```
procedure Set_Active(Combo_Box : access Gtk_Combo_Box_Record;
                      Index : Gint);
function Get_Active(Combo_Box : access Gtk_Combo_Box_Record)
return Gint;
function Get_Active_Text(Combo_Box : access Gtk_Combo_Box_Record)
return String;
```

Menu déroulant avec saisie

Il est également possible d'autoriser l'utilisateur à saisir une valeur non prédefinie. Reprenons l'exemple du programme présenté dans la Fiche d'identité. L'utilisateur ne peut utiliser le rouble, la livre sterling ou la roupie ; il est limité aux quatre choix proposés : Euro, Dollar, Yen et Yuan. Si vous souhaitez laisser davantage de liberté, il faudra utiliser un widget fils : les `GTK_Combo_Box_Entry`. Ceux-ci n'apportent aucune méthode supplémentaire et utilisent le même signal.

Sur la figure suivante, la `GTK_Combo_Box_Entry` permet d'entrer une valeur supplémentaire.



Un menu déroulant avec saisie

En résumé :

- Pour créer une barre d'icônes ou de menu, vous devez tout d'abord utiliser un premier widget comme conteneur : `GTK_Toolbar` ou `GTK_Menu_Bar`.
- Une fois la barre créée, ajoutez-y des items : `GTK_Menu_Item` pour la barre de menus et `GTK_Tool_Button` pour la barre d'icônes.
- Les `GTK_Menu_Item` ne sont en définitive que du texte. Pour qu'ils déroulent un menu, vous devrez utiliser les `GTK_Menu`. La même chose est réalisable avec la barre d'icônes en utilisant les `Gtk_Menu_Tool_Button` au lieu des `Gtk_Tool_Button`.
- Les widgets présentés dans ce chapitre sont des conteneurs. Il est donc possible de les imbriquer avec d'autres conteneurs afin de combiner divers widgets et d'obtenir des barres moins classiques.

[TP] Démineur (le retour)

Le démineur est de retour, pour votre plus grand malheur ! Mais cette fois, nous allons le peaufiner, le signoler et surtout l'achever ! À la fin de ce TP, vous aurez créé un véritable jeu complet et présentable. Prêts ? Alors ne perdons pas de temps et voyons tout de suite le cahier des charges.

Cahier des charges

Objectifs

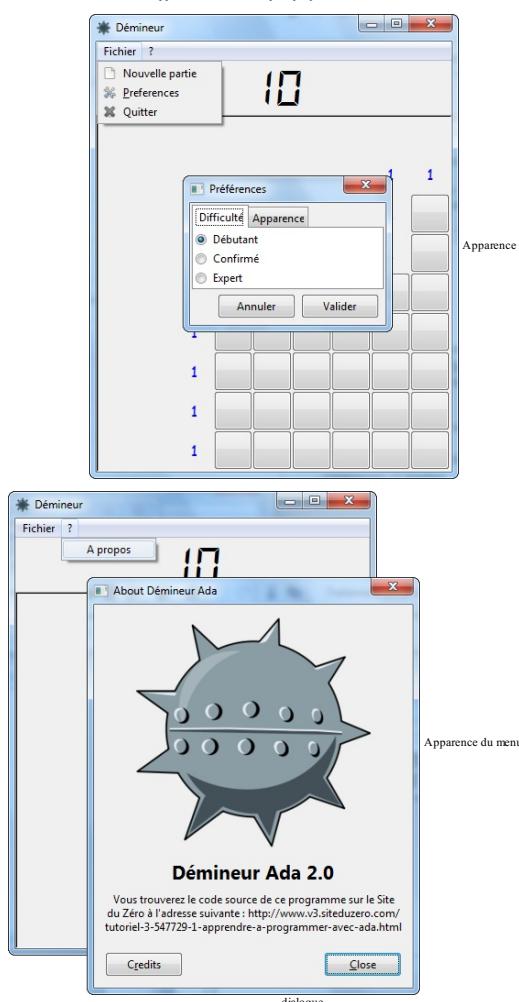
Vous aurez remarqué, la fois précédente, que notre démineur était encore limité. Nous ne pouvions faire qu'une seule partie, après quoi il fallait relancer le programme. Qui plus est, la taille de la grille et le nombre de bombes devaient être entrés à la main dans la console, ce qui gâchait quelque peu le plaisir d'avoir réalisé un jeu avecGtk. Votre mission, si vous l'acceptez, sera donc de combler ces lacunes :

1. Permettre au joueur de recommencer une partie après une victoire ou une défaite ou lorsqu'il en a assez de la partie en cours. Attention, il ne s'agit pas de supprimer la fenêtre pour en créer une nouvelle ! Il faudra seulement la réinitialiser.
2. Fournir des grilles prédéfinies selon des niveaux de difficulté. Ainsi, le niveau facile proposera des grilles de **16 × 16** cases comprenant 40 bombes ; le niveau difficile proposera des grilles de **16 × 30** cases comprenant 99 bombes.
3. Permettre au joueur de modifier l'apparence du jeu. Vous proposerez plusieurs types de drapeaux et plusieurs types de mines (un choix entre 2 images différentes suffira et pourra se limiter aux couleurs des drapeaux ou des mines). Ces options (niveau de difficulté et apparence) constitueront la principale difficulté.
4. Fournir au joueur une information sur le concepteur du jeu, sa version, etc.
5. Afficher proprement la grille de jeu. Avec les images fournies, les boutons ont besoin de 43 pixels pour s'afficher correctement. Dimensionnez la fenêtre en conséquence et pensez à inclure des barres de défilement si la grille est trop grande pour être affichée.
6. Enfin, permettre au joueur de quitter «proprement» le jeu et plus seulement en supprimant la console.

Ces objectifs sont simples et pourront être complétés à votre convenance (des idées d'améliorations seront proposées en conclusion, comme toujours).

Widgets nécessaires

Je vous laisse libre de donner à votre jeu l'apparence que vous désirez. Toutefois, vous devrez utiliser au moins une barre de menu ou une barre d'icônes. Je vous impose également l'utilisation de boutons radio pour choisir vos options ainsi que d'une barre d'onglets. De plus, comme dit précédemment, il faudra prévoir une barre de défilement au cas où il serait impossible d'afficher toutes les cases. Voici l'apparence, assez classique, que j'ai choisie :



Apparence du menu et de la boîte de dialogue

L'apparence choisie est simple : j'ai ajouté une barre de menu à la grille. Celle-ci comporte deux items : «Fichier» et «?». Le premier donne accès aux items «Nouvelle Partie», «Préférences» et «Quitter». Le second donne accès à l'item «À Propos». Par conséquent, il a fallu créer deux boîtes de dialogue supplémentaires : une boîte personnalisée pour paramétriser les options et une boîte de dialogue «À Propos».

Mais vous pouvez tout à fait choisir une apparence distincte. Voici une idée : utiliser une barre d'icônes pour les fonctionnalités basiques («nouvelle partie», «quitter», «à propos») et une barre d'onglets pour afficher soit la grille soit les options.

Une solution possible

Les spécifications

Les packages `P_Tile` et `P_Tile.Tile_Array` n'ont subi que des changements mineurs :

Code : Ada - P_Tile.ads

```
-- DEMINEUR --
-- P_Tile --
-- --
-- AUTEUR : KAJI9 --
-- DATE : 09/08/2013 --
-- --
```

```

--Ce package gère les cases de la grille. Il définit les
types--  --T_Tile_Record et T_File ainsi que les programmes
necessaires--  --pour initialiser, modifier ou détruire une case. --
--La variable globale Drapeaux_ restants y est déclarée
ainsi --  --que le type T_Status indiquant l'état d'une case. --
-----

WITH Gtk.Image ;           USE Gtk.Image ;
WITH Gtk.Button ;          USE Gtk.Button ;
WITH Gtk.Label ;           USE Gtk.Label ;

PACKAGE P_Tile IS

-----  -- TYPES --
-----

TYPE T_Status IS (Normal, Flag, Dug) ;
--Indique l'état d'une case :
-- Normal : la case existe encore et ne porte pas de drapeau
-- Flag : la case porte un drapeau
-- Dug : la case a été creusée, le bouton n'existe plus

TYPE T_Tile_Record IS TAGGED RECORD
Btn    : GTK_Button ;
Img    : Gtk.Image ;
Txt    : Gtk_Label ;
Mine   : Boolean := false ;
Nb     : Integer := 0 ;
Status : T_Status := Normal ;
END RECORD ;

TYPE T_Tile IS ACCESS ALL T_Tile_Record ;
--Les types permettant de manipuler les cases de la grille
-- Btn, Img, Txt sont les widgets correspondants
-- Mine indique si la case est minée
-- Nb indique le nombre de bombes alentours
-- Status indique l'état de la case

-----  -- VARIABLE GLOBALE --
-----

Drapeaux_Restants : Integer ;
--Permet le décompte des drapeaux utilisés et donc des bombes
découvertes

-----  -- PROGRAMMES --
-----

PROCEDURE Init_Tile (T : IN OUT T_Tile) ;
--Initialise la case

PROCEDURE Change_State(T : ACCESS T_Tile_Record'Class) ;
--Change l'état d'une case de Normal à Flag ou inversement

PROCEDURE Destroy (T : ACCESS T_Tile_Record'Class) ;
--Détruit le bouton de la case, Change son statut et charge
l'image ou le texte à afficher

PRIVATE

FUNCTION Set_Text (N : Integer) RETURN String ;
--Définit le texte à afficher sur une case ainsi que sa
couleur,
--N est le nombre à afficher

END P_Tile ;

```

Code : Ada - P_Tile-file_array.ads

```

-----  -- DEMINEUR --
-- P_Tile.Tile_Array --
-- 
-- AUTEUR : KAJI9 --
-- DATE : 09/08/2013 --
-- 
--Ce package gère les tableaux de T_Tile (cf package
P_Tile) -- Il définit le type T_Tile_Array ainsi que les programmes
-- 
--pour initialiser le tableau et pour tester si le joueur
a -- --gagné. --
-- 

-----  -- TYPES --
-----

TYPE T_Tile_Array IS ARRAY(integer range <>, integer range <>) OF
T_Tile ;
TYPE T_Tile_Array_Access IS ACCESS T_Tile_Array ;

-----  -- PROGRAMMES --
-----

PROCEDURE Init_Tile_Array(T : IN OUT T_Tile_Array ;
Width,Height,Bombs : Integer) ;
--Init_Tile_Array() permet de créer un tableau complet
ainsi que de placer aléatoirement
--des mines et d'affecter à chaque case le nombre de mines
alentour.
-- Width : largeur de la grille
-- Height : hauteur de la grille
-- Bombs : nombre de bombes

FUNCTION Victory(T : IN T_Tile_Array) RETURN Boolean ;
--Victory() Renvoie TRUE si toutes les cases non minées ont
été découvertes, et
--FALSE s'il reste des cases à creuser

PRIVATE

PROCEDURE Increase(T : IN OUT T_Tile_Array ; X,Y : Integer) ;
--Increase() permet d'augmenter le nombre de bombes connues
d'une case
--de 1 point. X et Y sont les coordonnées de la bombe.

END P_Tile.Tile_Array ;

```

En revanche le package Main.Window a disparu. Il devait être le package principal, centralisant tous les autres, mais le type T_Game étant de plus en plus utilisé par les autres packages, il est devenu nécessaire de revoir l'architecture. Il est désormais scindé en deux : P_Game qui définit notamment le type T_Game et est utilisé par quasiment tous les autres packages ; et P_Game.Methods qui définit les méthodes nécessaires au fonctionnement du jeu et centralisent toutes les données.

Vous noterez également que le type T_Game_Record a été très largement complété pour intégrer les nouveaux widgets, voire même modifié pour permettre la réinitialisation de certains paramètres. Ainsi, le paramètre Tab qui était un tableau est devenu un pointeur sur tableau.

Code : Ada - P_Game.ads

```

-----  -- DEMINEUR --
-- P_Game --
-- 
-- AUTEUR : KAJI9 --
-- DATE : 09/08/2013 --
-- 
```

```

-- --
-- Ce package définit les types T_Game_Record et T_Game qui
-- contiennent les informations liées à la partie, notamment
la -- --grille de cases ou les principaux widgets. --
et -- --Il définit également les types T_Option, T_Option_Access
fils -- --T_Niveau ainsi que la variable globale Title. --
les -- --Les méthodes associées ont été reléguées dans un package
    --appelé T_Game.Methods ce qui évite tout redondance dans
    --appels de packages. --
-----

WITH Gtk.Window ;           USE Gtk.Window ;
WITH Gtk.Table;             USE Gtk.Table;
WITH Gtk.Widget;            USE Gtk.Widget;
WITH Gtk.Box;               USE Gtk.Box;
WITH Gtk.Label;              USE Gtk.Label;
WITH Gtk.Menu_Bar;          USE Gtk.Menu_Bar;
WITH Gtk.Menu_Item;          USE Gtk.Menu_Item;
WITH Gtk.Image_Menu_Item;   USE Gtk.Image_Menu_Item;
WITH Gtk.Menu;               USE Gtk.Menu;
WITH Gtk.Scrolled_Window;   USE Gtk.Scrolled_Window ;
WITH P.Tile;                USE P.Tile ;
WITH P.Tile.Tile_Array ;    USE P.Tile.Tile_Array ;

PACKAGE P_Game IS
-----  

-- VARIABLE GLOBALE --
-----  

  Title : CONSTANT String := "Démineur" ;
  --Titre du jeu  

-----  

-- TYPES --
-----  

  TYPE T_Option IS ARRAY(1..3) OF Integer ;
  --Ce type permet d'enregistrer facilement les choix de
  l'utilisateur quant à
  -- la difficulté ou aux graphismes lors de l'utilisation de
  boîte de dialogue.
  --case n°1 : niveau de difficulté
  --case n°2 : nom du fichier pour les drapeaux
  --case n°3 : nom du fichier pour les mines
  TYPE T_Option_Access IS ACCESS T_Option ;
  TYPE T_Niveau IS (Facile, Medium, Difficile) ;
  --Type utilisé pour définir le niveau de difficulté plus
  clairement qu'avec T_Option
  TYPE T_Game_Record IS RECORD
    Tab : T.Tile_Array_Access ;
    X,Y : Integer ;
    Win : Gtk.Window;
    Grille : Gtk.Table ;
    Compteur : Gtk.Label ;
    Box : Gtk.Vbox ;
    Barre : Gtk.Menu_Bar ;
    Item_Fichier : Gtk.Menu_Item ;
    Item_Question : Gtk.Image_Menu_Item ;
    Menus_Fichier : Gtk.Menu ;
    Menus_Question : Gtk.Menu ;
    Item_A_Propos : Gtk.Menu_Item ;
    Item_Nouveau : Gtk.Image_Menu_Item ;
    Item_Option : Gtk.Image_Menu_Item ;
    Item_Quitter : Gtk.Image_Menu_Item ;
    Ascenseur : Gtk.Scrolled_Window ;
    Niveau : T_Niveau := Facile ;
    Width : Integer := 9 ;
    Height : Integer := 9 ;
    Bombs : Integer := 10 ;
    General_Option : T_Option := (1,1,1) ;
  END RECORD ;
  TYPE T_Game IS ACCESS ALL T_Game_Record ;
  --Contiennent la plupart des informations sur la partie :
  -- Width : largeur de la grille de cases
  -- Height : hauteur de la grille de cases
  -- Bombs : Nombre de bombes
  -- General_Option : Tableau résumant les choix d'options de
  l'utilisateur
  -- Tab : grille de cases
  -- X, Y : variables permettant de transmettre les coordonnées
  de la case cliquée
  -- Win : widget fenêtre du jeu
  -- Grille : widget GTK Table contenant tous les boutons
  -- Compteur : widget affichant la variable globale
  Drapeaux_restants
  -- Box : widget contenant Compteur et Box
  -- Ascenseur: widget affichant les barres de défilement
  (seulement si nécessaire)
  -- Items et Menus de la barre de menus
END P_Game ;

```

Code : Ada - P_Game-Methods.ads

```

-- DEMINEUR --
-- P_Game.Methods --
-- --
-- AUTEUR : KAJI9 --
-- DATE : 09/08/2013 --
-- --
-- Ce package définit les fonctions de callback Click_on, -
-- Exit_window et les procédures d'initialisation ou de --
-- réinitialisation des types T_Game_Record et T_Game. --
-- La décomposition en deux packages évite la redondance
des -- --appels de packages. --
-----  

WITH Gtk.Button ;           USE Gtk.Button ;
WITH Gtk.Handlers ;         USE Gtk.Handlers ;
WITH Gdk.Event ;            USE Gdk.Event ;
WITH Gtk.Widget ;            USE Gtk.Widget ;
WITH P_Game ;                USE P_Game ;

PACKAGE P_Game_Methods IS
-----  

-- PACKAGES --
-----  

  PACKAGE P_Simple_Callback IS NEW
    Gtk.Handlers.Callback(Gtk.widget_Record) ;
    USE P_Simple_Callback ;
    --Package pour la fermeture de la fenêtre
  PACKAGE P_Callback IS NEW
    Gtk.Handlers.User_Callback(Gtk.widget_Record, T_Game) ;
    USE P_Callback ;
    --Package pour les menus
  PACKAGE P_Return_Callback IS NEW
    Gtk.Handlers.User_Return_Callback(Gtk_Button_Record, boolean,
    T_Game_Record) ;
    USE P_Return_Callback ;
    --Package pour les callbacks liés aux boutons de la grille
-----  

-- PROGRAMMES --
-----  


```

```

PROCEDURE Init_Game(Game : T_Game) ;
  --Procédure d'initialisation du jeu
  --Les paramètres correspondent à ceux du type T_Game_Record
FUNCTION click_on(Emetteur : ACCESS Gtk_Button_Record'Class ;
  Eventement : Gdk_Event) RETURN Boolean;
  --Callback appelé lorsque le joueur clique sur un bouton
  --Celui-ci permet de placer ou d'enlever un drapeau
  --mais aussi de creuser une case
PROCEDURE Exit_Window (Emetteur : Access
  GTK_Widget_Record'Class) ;
  --Callback appelé pour fermer la fenêtre
PROCEDURE Reinit_Game (Game : T_Game) ;
  --Réinitialise le jeu sans détruire la fenêtre et en tenant
compte
  --des paramètres définis par l'utilisateur
PRIVATE
  PROCEDURE Init_Window (Game : T_Game ; Width,Height : Integer)
  ; --initialise la fenêtre de jeu
  PROCEDURE Init_Compteur (Game : T_Game) ;
  --initialise le compteur
  PROCEDURE Init_Menu (Game : T_Game) ;
  --initialise le menu
  PROCEDURE Init_Box (Game : T_Game) ;
  --initialise le paramètre Box et y ajoute les widgets
Compteur et Grille
  PROCEDURE Init_Grille (Game : T_Game) ;
  --initialise la grille de boutons et connecte à chacun son
callback
  PROCEDURE Set_Compteur (Game : T_Game_Record) ;
  --met à jour le compteur de drapeaux
  PROCEDURE Explosion (Game : T_Game_Record ; X,Y : Integer) ;
  --affiche la bombe et lance la boîte de dialogue de défaité
  PROCEDURE Creuser_Autour (Game : T_Game_Record ; X,Y : Integer) ;
  --détruit les 8 cases entourant la case de coordonnées (X,Y)
  --appelle la procédure Creuser si l'une des 8 cases est
nulle
  PROCEDURE Creuser (Game : T_Game_Record ; X,Y : Integer) ;
  --détruit la case de coordonnées (X,Y). Lance explosion si la
case
  --est minée ; lance Creuser autour si la case est nulle
  PROCEDURE Reinit_Game (Emetteur : ACCESS
  GTK_Widget_Record'Class) ;
  --Callback appellé pour réinitialiser la fenêtre. Se contente
d'appeler
  --la procédure publique du même nom
END P_Game.Methods ;

```

Le package P_Dialog a logiquement été complété, et même lourdement complété. La boîte de dialogue Option_Dialog appartenant de nombreuses exigences avec elle, il a fallu ajouter un package P_Option pour gérer les options. Ainsi, les widgets modifient un tableau d'entiers appelé T_Option qui est ensuite réinterprété pour connaître le niveau de difficulté ou le nom des fichiers utilisés.

Code : Ada - P_Option.ads

```

-----  

-- DEMINEUR --  

-- P_Option --  

-- AUTEUR : KAJI9 --  

-- DATE : 09/08/2013 --  

-- --  

-- Ce package définit les principales méthodes pour  

connaitre ou --  

  --mettre à jour les options : niveau de difficulté,  

fichier --  

  --choisi pour les mines ou pour les drapeaux. --  

  --Il est principalement utilisé par P_Dialog et la boîte  

de --  

  --dialogue Option_Dialog. --  

-----  

WITH Ada.Strings.Unbounded ; USE Ada.Strings.Unbounded ;
WITH P_Game ; USE P_Game ;  

  
PACKAGE P_Option IS  

-----  

-- VARIABLES GLOBALES --  

-----  

  Mine_Filename : Unbounded_String := To_unbounded_string("mine-
rouge.png") ;
  Drapeau_Filename : Unbounded_String :=
  To_Unbounded_String("drapeau-bleu.png") ;
  Current_Option : T_Option := (1,1,1) ;  

-----  

-- METHODES --  

-----  

  PROCEDURE Set_Difficulty(game : T_Game) ;
  PROCEDURE Set_Drapeau_Filename(Option : T_Option) ;
  PROCEDURE Set_Mine_Filename(Option : T_Option) ;
  --Ces procédures ajustent les paramètres du jeu
  --(nombre de bombes, tailles de la grille, nom
  --des fichiers en fonction des options choisies)  

PRIVATE
  FUNCTION Get_Difficulty(Option : T_Option) RETURN T_Niveau ;
  FUNCTION Get_Drapeau_Filename(Option : T_Option) RETURN String ;
  FUNCTION Get_Mine_Filename(Option : T_Option) RETURN String ;
  --Ces fonctions transforment un tableau d'entiers (T_Option)
  --en valeurs plus aisément lisibles et utilisables comme une
  --chaîne de caractères pour les noms des fichiers ou un type
  --T_Niveau dont les valeurs sont compréhensibles : Facile,
Medium
  -- et Difficile.
END P_Option ;

```

Code : Ada - P_Dialog.ads

```

-----  

-- DEMINEUR --  

-- P_Dialog --  

-- --  

-- AUTEUR : KAJI9 --  

-- DATE : 09/08/2013 --  

-- --  

-- Ce package définit les quatre boîtes de dialogue du jeu
: --  

  --Loose_Dialog qui s'ouvre si vous cliquez sur une mine ;
--  

  --Win_Dialog qui s'ouvre si vous avez découvert toutes les
--  

  --mines ; --  

  --About_Dialog qui s'ouvre lorsque vous cliquez sur A
propos --
  --Option_Dialog qui s'ouvre lorsque vous cliquez sur
Préférences --
  --Il fournit également les procédures nécessaires à leur --
  --initialisation --
-----  

WITH Gtk.Widget ; USE Gtk.Widget ;
WITH Gtk.Dialog ; USE Gtk.Dialog ;
WITH Gtk.Message_Dialog ; USE Gtk.Message_Dialog ;
WITH Gtk.About_Dialog ; USE Gtk.About_Dialog ;
WITH Gtk.Window ; USE Gtk.Window ;
WITH Gtk.Enums ; USE Gtk.Enums ;

```

```

WITH P_Game ;           USE P_Game ;
WITH Gtk.Handlers ; 

PACKAGE P_Dialog IS
    -- VARIABLES GLOBALES --
    ----

    Loose_Dialog : Gtk_Message_Dialog ;
    Win_Dialog : Gtk_Message_Dialog ;
    About_Dialog : Gtk_About_Dialog ;
    Option_Dialog : Gtk_Dialog ;

    ----
    -- PROGRAMMES --
    ----

    PROCEDURE Init_Loose_Dialog(Parent : Gtk_Window) ;
    PROCEDURE Init_Win_Dialog (Parent : Gtk_Window) ;
    PROCEDURE Init_About_Dialog ;
    PROCEDURE Init_Option_Dialog(Game : T_Game) ;
    --Initialisent les boîtes dialogues ci-dessus
    --et les indiquent à la fenêtre mère (Game.Win)
    PROCEDURE Run_About_Dialog(Emetteur : ACCESS
        Gtk_Widget_Record'Class) ;
    PROCEDURE Run_Option_Dialog(Emetteur : ACCESS
        Gtk_Widget_Record'Class) ;
        Game : T_Game) ;
    --Callbacks appelés pour lancer les boîtes de dialogue
    About_Dialog et Option_Dialog

PRIVATE
    PACKAGE P_Callback IS NEW
        Gtk.Handlers'User_Callback(Gtk_Widget_Record, Integer) ;
    USE P_Callback ;

    PROCEDURE Change_Difficulte(Emetteur : ACCESS
        Gtk_Widget_Record'Class ;          Valeur : Integer) ;
    PROCEDURE Change_Drapeau_Filename(Emetteur : ACCESS
        Gtk_Widget_Record'Class ;          Valeur : Integer) ;
    PROCEDURE Change_Mine_Filename(Emetteur : ACCESS
        Gtk_Widget_Record'Class ;          Valeur : Integer) ;
    --Callbacks utilisés par la boîte de dialogue Option_Dialog :
    --ils permettent de connaître le choix fait par l'utilisateur
    --en termes de difficulté ou d'image pour les drapeaux et les
    --mines.
END P_Dialog ;

```

Le corps des packages

Vérons-en maintenant au corps de ces packages. Attention, il n'existe pas de fichier P_Game.adb, ce package ne faisant que définir des types et variables.

Code : Ada - P_Tile.adb

```

-----  

-- DEMINEUR --  

-- P_Tile --  

-- --  

-- AUTEUR : KAJI9 --  

-- DATE : 09/08/2013 --  

-- --  

-- Ce package gère les cases de la grille. Il définit les  

types--  

--T_Tile_Record et T_Tile ainsi que les programmes  

necessaires--  

--pour initialiser, modifier ou détruire une case. --  

--La variable globale Drapeaux_restants y est déclarée  

ainsi --  

--que le type T_Status indiquant l'état d'une case. --  

-----  

-----  

WITH P_Option ;           USE P_Option ;
WITH Ada.Strings.Unbounded ; USE Ada.Strings.Unbounded ;

PACKAGE BODY P_Tile IS
    PROCEDURE Init_Tile(T : in out T_Tile) IS
    BEGIN
        T := new T_Tile_record ;
        GTK_new(T.Btn) ;
        T.Mine := False ;
        T.Nb := 0 ;
        T.Status := normal ;
        END Init_Tile ;

    PROCEDURE Change_State(T : ACCESS T_Tile_Record'Class) IS
    BEGIN
        IF T.Status = Normal THEN
            T.Status := Flag ;
            GTK_New(T.Img, "./" & to_string(Drapeau_Filename)) ;
            T.Btn.Add(T.Img) ;
            T.Img.Show ;
            Drapeaux_restants := Drapeaux_restants - 1 ;
        ELSE
            T.Status := Normal ;
            T.Img.Destroy ;
            Drapeaux_restants := Drapeaux_restants + 1 ;
        END IF ;
    END Change_State ;

    FUNCTION Set_Text(N : Integer) RETURN String IS
    BEGIN
        CASE N IS
            WHEN 1 => RETURN "<span font_desc='comic sans ms 12' foreground='blue'>1</span>" ;
            WHEN 2 => RETURN "<span font_desc='comic sans ms 12' foreground='#09GA09'>2</span>" ;
            WHEN 3 => RETURN "<span font_desc='comic sans ms 12' foreground='red'>3</span>" ;
            WHEN 4 => RETURN "<span font_desc='comic sans ms 12' foreground='#003399'>4</span>" ;
            WHEN 5 => RETURN "<span font_desc='comic sans ms 12' foreground='#6C0277'>5</span>" ;
            WHEN 6 => RETURN "<span font_desc='comic sans ms 12' foreground='#009999'>6</span>" ;
            WHEN 7 => RETURN "<span font_desc='comic sans ms 12' foreground='#D9EAD1'>7</span>" ;
            WHEN 8 => RETURN "<span font_desc='comic sans ms 12' foreground='#606060'>8</span>" ;
            WHEN OTHERS => RETURN "" ;
        END CASE ;
    END Set_Text ;

    PROCEDURE Destroy(T : ACCESS T_Tile_Record'Class) IS
    BEGIN
        IF T.Status = Normal THEN
            Destroy(T.Btn) ;
            IF T.Mine THEN
                GTK_New(T.Img, "./" & to_string(mine_filename)) ;
                T.Img.show ;
            ELSE
                Gtk_New(T.Txt, set_text(T.nb)) ;
                T.Txt.set_use_markup(true) ;
                T.Txt.Show ;
            END IF ;
        END IF ;
    END Destroy ;
END P_Tile ;

```

Code : Ada - P_Tile.Tile_array.adb

```

-----  

-- DEMINEUR --  

-- P_Tile.Tile_Array --  

-- --  

-- AUTEUR : KAJI9 --  

-- DATE : 09/08/2013 --  

-- --  

-- Ce package gère les tableaux de T_Tile (cf package  

P_Tile) -- Il définit le type T_Tile_Array ainsi que les programmes  

-- pour initialiser le tableau et pour tester si le joueur  

a -- a gagné.  

-----  

-----  

WITH Ada.Numerics.Discrete_Random ;  

PACKAGE BODY P_Tile.Tile_Array IS  

    PROCEDURE Init_Tile_Array(T : IN OUT T_Tile_Array ;  

width,height,bombs : integer) IS  

        subtype random_range is integer range 1..width*height ;  

    PACKAGE P_Random IS NEW  

Ada.Numerics.Discrete_Random(Random_Range) ;  

    USE P_Random ;  

    G : Generator ;  

    X,Y : Integer ;  

    Reste : Integer := Bombs ;  

BEGIN  

    Reset(G) ;  

    --Création des cases  

    FOR J IN 1..height LOOP  

        FOR I IN 1..width LOOP  

            Init_Tile(T(I,J)) ;  

        END LOOP ;  

    END LOOP ;  

    --Placement aléatoire des bombes et calcul des nombres  

    associés à chaque case  

    WHILE Reste > 0 LOOP  

        X := Random(G) mod Width + 1 ;  

        Y := Random(G) mod Height + 1 ;  

        IF T(X,Y).Mine = false  

        THEN T(X,Y).Mine:=True ;  

        Increase(T,X,Y) ;  

        Reste := Reste - 1 ;  

    END IF ;  

    END LOOP ;  

END Init_Tile_Array ;  

    PROCEDURE Increase(T : IN OUT T_Tile_Array ; X,Y : Integer) IS  

        xmin,xmax,ymin,ymax : integer ;  

BEGIN  

    xmin := integer'max(1 , x-1) ;  

    xmax := integer'min(x+1 , T'last(1)) ;  

    ymin := integer'max(1 , Y-1) ;  

    ymax := Integer'Min(Y+1, T'Last(2)) ;  

    FOR J IN ymin..ymax LOOP  

        FOR I IN xmin..xmax LOOP  

            T(I,J).Nb := T(I,J).Nb + 1 ;  

        END LOOP ;  

    END LOOP ;  

    END Increase ;  

    FUNCTION Victory(T : IN T_Tile_Array) RETURN Boolean IS  

        Nb_mines,Nb_cases : integer := 0 ;  

BEGIN  

    --Décompte du nombre de mines et de cases non détruites  

    FOR J IN T'RANGE(2) LOOP  

        FOR I IN T'RANGE(1) LOOP  

            IF T(i,j).Status = normal OR t(i,j).status = flag  

            THEN nb_cases := nb_cases + 1 ;  

        END IF ;  

        IF T(i,J).Mine  

        THEN Nb_Mines := Nb_Mines + 1 ;  

        END IF ;  

    END LOOP ;  

    --Renvoi du résultat  

    RETURN Nb_Mines = Nb_Cases ;  

END Victory ;  

END P_Tile.Tile_Array ;

```

Code : Ada - P_Game.Methods.adb

```

-----  

-- DEMINEUR --  

-- Main_Window --  

-- --  

-- AUTEUR : KAJI9 --  

-- DATE : 17/06/2013 --  

-- --  

-- Ce package définit les types T_Game_Record et T_Game qui  

-- contiennent les informations liées à la partie, notamment  

la -- grille de cases ou les principaux widgets. Il définit  

aussi -- la fonction de callback (Click_on) et la procédure --  

--d'initialisation.  

-----  

-----  

WITH Glib ; USE Glib ;  

WITH Glib.Convert ; USE Glib.Convert ;  

WITH Gtk.Window ; USE Gtk.Window ;  

WITH Gtk.Table ; USE Gtk.Table ;  

WITH Gtk.Box ; USE Gtk.Box ;  

WITH Gtk.Label ; USE Gtk.Label ;  

WITH Gtk.Menu_Bar ; USE Gtk.Menu_Bar ;  

WITH Gtk.Menu_Item ; USE Gtk.Menu_Item ;  

WITH Gtk.Image_Menu_Item ; USE Gtk.Image_Menu_Item ;  

WITH Gtk.Menu ; USE Gtk.Menu ;  

WITH Gtk.Scrolled_Window ; USE Gtk.Scrolled_Window ;  

WITH Gtk.Dialog ; USE Gtk.Dialog ;  

WITH Gtk.Menubar ; USE Gtk.Menubar ;  

WITH Gtk.Enums ; USE Gtk.Enums ;  

WITH P_Tile ; USE P_Tile ;  

WITH P_Tile.Tile_Array ; USE P_Tile.Tile_Array ;  

WITH P_Dialog ; USE P_Dialog ;  

WITH Ada.Unchecked_Deallocation ;  

PACKAGE BODY P_Game.Methods IS  

    PROCEDURE Init_Window(Game : T_Game ; Width,Height : Integer) IS  

    BEGIN  

        --Création d'une fenêtre avec callback et dimensions  

correctes  

        Gtk_New(Game.Win) ;  

        Game.Win.Set_Default_Size(43*Gint(width),82 + 43*Gint(Height))  

;  

        Game.Win.Set_Title(Locale_To_Utf8>Title) ;  

        IF Game.Win.Set_Icon_From_File("mine-noire.png")  

        THEN NULL ;  

        END IF ;  

        connect(game.win, "destroy", exit_window'access) ;  

    END Init_Window ;  

    PROCEDURE Init_Compteur(Game : T_Game) IS  

    BEGIN  

        Gtk_New(Game.Compteur,"<span font_desc='DS-Digital 45'>" &  

Integer'image(drapeaux restants) & "</span>" ) ;  

        Game.Compteur.Set_Use_Markup(True) ;

```

```

END Init_Compteur ;

PROCEDURE Init_Menu      (Game : T_Game) IS
BEGIN
    --Création des menus et des items
    Gtk_New(Game.Barre) ;
    Gtk_New(Game.Item_Fichier, "Fichier") ;
    Gtk_New(Game.Item_Question, "?") ;
    Gtk_New_From_Stock(Game.Item_Nouveau, "gtk-new") ;
    Game.Item_Nouveau.set_label("Nouvelle partie") ;
    Gtk_New_From_Stock(Game.Item_Option, "gtk-preferences") ;
    Gtk_New_From_Stock(Game.Item_Qitter, "gtk-close") ;
    Game.Item_Qitter.set_label("Quitter") ;
    Ctk_New(Ctk.Item_A_Propos, "A propos") ;
    Gtk_New(Game.Menu_Fichier) ;
    Gtk_New(Game.Menu_Questions) ;
    --Organisation des menus
    Game.Barre.Append(Game.Item_Fichier) ;
    Game.Item_Fichier.Set_Submenu(Game.Menu_Fichier) ;
    Game.Barre.Append(Game.Item_Question) ;
    Game.Item_Question.Set_Submenu(Game.Menu_Questions) ;
    Game.Menu_Fichier.Append(Game.Item_Nouveau) ;
    Game.Menu_Fichier.Append(Game.Item_Option) ;
    Game.Menu_Fichier.Append(Game.Item_Qitter) ;
    Game.Menu_Questions.Append(Game.Item_A_Propos) ;
    --Connexions avec les callbacks appropriés
    Connect(Game.Item_Qitter, "activate", Exit_Window'ACCESS) ;
    Connect(Game.Item_A_Propos, "activate", Run_About_Dialog'ACCESS) ;
    Connect(Game.Item_Option, "activate", Run_Option_Dialog'ACCESS, Game) ;
    Connect(Game.Item_Nouveau, "activate", Reinit_Game'ACCESS, Game) ;
END Init_Menu ;

PROCEDURE Init_Box(Game : T_Game) IS
BEGIN
    --Organisation de la fenêtre avec boîtes et ascenseurs
    Gtk_New_Vbox(Game.Box) ;
    Game.Box.Pack_Start(Game.Barre,      Expand => False, Fill=>
True) ;
    Game.Box.Pack_Start(Game.Compteur,  Expand => False, Fill =>
False) ;
    Gtk_New(Game.Ascenseur) ;
    Game.Ascenseur.Set_Policy(Policy_Automatic, Policy_Automatic)
;
    Game.box.pack_start(Game.Ascenseur, Expand => true,  Fill =>
true) ;
    Game.Ascenseur.add_with_viewport(Game.Grille) ;
    Game.Win.Add(Game.Box) ;
END Init_Box ;

PROCEDURE Init_Grille(Game : T_Game) IS
BEGIN
    --Création de la GTK_Table et du T_Tile_Array
    Gtk_New(Game.Grille) ;
    Guint(Game.Width),          Guint(Game.Height),
    True) ;
    Init_Tile_Array(Game.Tab.all,
                    Game.Width,
                    Game.Height,
                    Game.Bombs) ;
    --Implantation des différents boutons et connexion de
    --chacun avec son callback
    FOR J IN 1..game.height LOOP
        FOR I IN 1..game.width LOOP
            Game.Grille.Attach(Game.Tab(I,J).Btn,
                               Guint(I)-1,Guint(I),
                               Guint(J)-1,Guint(J)) ;
            Game.X := I ;
            Game.Y := J ;
            Connect(Game.Tab(I,J).Btn,
                    Signal_Button_Press_Event,
                    To_Marshaller(click_on'ACCESS),
                    Game.all) ;
        END LOOP ;
    END LOOP ;
END Init_Grille ;

PROCEDURE Reinit_Game(Game      : T_Game) IS
    PROCEDURE Free IS NEW
    Ada.Unchecked_Deallocation(T_Tile_Array,T_Tile_Array_Access) ;
BEGIN
    --Réinitialisation d'une fenêtre : destruction et
    --recréation
    --de certains objets
    Free(Game.Tab) ;
    Game.Tab := new T_Tile_Array(1..game.width, 1..game.height) ;
    Game.Win.Resize(43*Gint(game.Width), 82 +
43*Gint(game.Height)) ;
    Drapeaux_Restants := game.Bombs ;
    Set_compteur(Game.all) ;
    Game.Grille.Destroy ;
    Init_Grille(Game) ;
    Game.Ascenseur.add_with_viewport(Game.Grille) ;
    Game.Win.Show_All ;
    END Reinit_Game ;

PROCEDURE Reinit_Game(Emetteur : ACCESS Gtk.widget_Record'Class ;
                     Game      : T_Game) IS
    PRAGMA Unreferenced(Emetteur) ;
BEGIN
    Reinit_Game(Game) ;
    END Reinit_Game ;

PROCEDURE Init_Game(Game : T_Game) IS
BEGIN
    Game.Tab := new T_Tile_Array(1..game.width, 1..game.height) ;
    Init_Window(Game,game.Width,game.height) ;
    Init_Compteur(Game) ;
    Init_menu(Game) ;
    Init_Grille(Game) ;
    Init_Box(Game) ;
    Game.Win.Show_All ;
    Init_Loose_Dialog(Game.Win) ;
    Init_Win_Dialog'(Game.Win) ;
END Init_Game ;

PROCEDURE Set_Compteur(Game : T_Game_Record) IS
BEGIN
    IF Drapeaux_Restants < 0
        THEN Game.Compteur.Set_Label("<span foreground = 'red'
font_desc='DS-Digital 45'>" &
Integer'image(Drapeaux_Restants) &
" </span>") ;
        ELSE Game.Compteur.Set_Label("<span foreground = 'black'
font_desc='DS-Digital 45'>" &
Integer'image(Drapeaux_Restants) &
" </span>") ;
    END IF ;
END Set_Compteur ;

FUNCTION click_on(Emetteur : ACCESS Gtk.Button_Record'Class ;
                  Evenement : Gdk_Event ;
                  Game      : T_Game_Record) RETURN Boolean IS
    X : CONSTANT Integer := Game.X ;
    Y : CONSTANT Integer := Game.Y ;
    PRAGMA Unreferenced(Emetteur) ;
BEGIN
    --Choix des procédures à lancer selon le bouton cliqué
    CASE Get_Button(Evenement) IS
        WHEN 1 => Creuser(Game,X,Y) ;
        WHEN 3 => Game.Tab(X,Y).change_state ;
        Set_Compteur(Game) ;
        WHEN OTHERS => NULL ;
    END CASE ;
END click_on ;

```

```
--Teste de victoire et lancement éventuels de la boîte de dialogue
--de victoire. Notes bien le "AND THEN"
IF Victory(Game.Tab.all) AND THEN Win_Dialog.Run =
  Gtk_Response.Ok
  THEN Game.Item.Nouveau.Activate ;
    Win.Dialog.Destroy ;
    Init_Win_Dialog(Game.Win) ;
END IF ;
RETURN False ;
END Click_On ;

PROCEDURE Exit_Window (Emetteur : ACCESS
  GTK_Widget_Record'Class) IS
  PRAGMA Unreferenced(Emetteur) ;
BEGIN
  Main_Quit ;
END Exit_Window ;

PROCEDURE Explosion(Game : T_Game_Record ; X,Y : Integer) IS
BEGIN
  --Affichage de l'image de la bombe cliquée
  Game.Grille.Attach(Game.Tab(x,y).Img,
    Guint(X)-1,
    Guint(X),
    Guint(Y)-1,
    Guint(Y)) ;
  --Ouverture de la boîte de dialogue de défaite
  IF Loose_Dialog.Run = Gtk_Response.Ok
  THEN Game.Item.Nouveau.Activate ;
    Loose_Dialog.Destroy ;
    Init_Loose_Dialog(Game.Win) ;
  END IF ;
END Explosion ;

PROCEDURE Creuser_Atour(Game : T_Game_Record ; X,Y : Integer) IS
Xmin,Xmax,Ymin,Ymax : Integer ;
tile : T_Tile ;
BEGIN
  Xmin := integer'max(1 , x-1) ;
  Xmax := integer'min(x+1 , Game.Tab'last(1)) ;
  Ymin := integer'max(1 , Y-1) ;
  Ymax := Integer'Min(Y+1 , Game.Tab'Last(2)) ;
  --parcoure les cases autour de (X,Y)
  FOR J IN Ymin..Ymax LOOP
    FOR I IN Xmin..Xmax LOOP
      Tile := Game.Tab(I,J) ;
      --si la case porte un chiffre, elle est simplement
      --détruite,
      --sinon, on lance un appel récursif via la procédure
      Creuser()
      IF Tile.status = Normal AND Tile.nb > 0
      THEN Tile.destroy ;
        Tile.Status := Dug ;
        Game.Grille.Attach(Tile.txt,
          Guint(I)-1,Guint(I),
          Guint(J)-1,Guint(J)) ;
      ELSIF Tile.Status = normal
      THEN Creuser(Game,I,J) ;
      END IF ;
    END LOOP ;
  END LOOP ;
END Creuser_Atour ;

PROCEDURE Creuser(Game : T_Game_Record ; X,Y : Integer) IS
  tile : CONSTANT T_Tile := Game.tab(x,y) ;
BEGIN
  Tile.destroy ;
  --Si la case est minée
  IF Tile.Status = Normal AND Tile.Mine
  THEN Explosion(Game,X,Y) ;
  --Si la case n'est ni minée ni creusée
  ELSIF Tile.Status = Normal
  THEN Tile.Status := Dug ;
    Game.Grille.Attach(Tile.txt,
      Guint(X)-1,Guint(X),
      Guint(Y)-1,Guint(Y)) ;
  --Si la case est nulle, on lance Creuser_around()
  IF Tile.Nb = 0
  THEN Creuser_around(Game,x,y) ;
  END IF ;
END Creuser ;
END P_Game.Methods ;
```

Code : Ada - P_Option.adb

```
-----  
-- DEMINEUR --  
-- P_Option --  
-- --  
-- AUTEUR : KAJI9 --  
-- DATE : 09/08/2013 --  
-- --  
-- Ce package définit les principales méthodes pour  
connaître ou --  
-- mettre à jour les options : niveau de difficulté,  
fichier --  
-- choisi pour les mines ou pour les drapeaux. --  
-- Il est principalement utilisé par P_Dialog et la boîte  
de --  
-- dialogue Option Dialog. --  
-----  
  
PACKAGE BODY P_Option IS  
  FUNCTION Get_Difficulty(Option : T_Option) RETURN T_Niveau IS
  BEGIN
    CASE Option(1) IS
      WHEN 1 => RETURN Facile ;
      WHEN 2 => RETURN Medium ;
      WHEN OTHERS => RETURN Difficile ;
    END CASE ;
  END Get_Difficulty ;  
  
  FUNCTION Get_Drapeau_Filename(Option : T_Option) RETURN String IS
  BEGIN
    CASE Option(2) IS
      WHEN 1 => RETURN "./drapeau-bleu.png" ;
      WHEN OTHERS => RETURN "./drapeau-rouge.png" ;
    END CASE ;
  END Get_Drapeau_Filename ;  
  
  FUNCTION Get_Mine_Filename(Option : T_Option) RETURN String IS
  BEGIN
    CASE Option(3) IS
      WHEN 1 => RETURN "./mine-rouge.png" ;
      WHEN OTHERS => RETURN "./mine-noire.png" ;
    END CASE ;
  END Get_Mine_Filename ;  
  
  PROCEDURE Set_Difficulty(game : T_Game) IS
  BEGIN
    Game.Niveau := Get_Difficulty(Game.General_Option) ;
    CASE game.Niveau IS
      WHEN Facile => game.Width := 9 ;
        game.Height := 9 ;
        game.Bombs := 10 ;
      WHEN Medium => game.Width := 16 ;
        game.Height := 16 ;
        game.Bombs := 40 ;
      WHEN Difficile => game.Width := 30 ;
        game.Height := 16 ;
        game.Bombs := 99 ;
    END CASE ;
  END Set_Difficulty ;
```

```

END Set_Difficulty ;

PROCEDURE Set_Drapeau_Filename(Option : T_Option) IS
BEGIN
  Drapeau_Filename :=
    To_unbounded_string(Get_Drapeau_Filename(Option)) ;
END Set_Drapeau_Filename ;

PROCEDURE Set_Mine_Filename(Option : T_Option) IS
BEGIN
  Mine_Filename :=
    To_unbounded_string(Get_Mine_Filename(Option)) ;
END Set_Mine_Filename ;

END P_Option ;

```

Code : Ada - P_Dialog.adb

```

-----  

-- DEMINEUR --  

-- P_Dialog --  

-- --  

-- AUTEUR : KAJI9 --  

-- DATE : 17/06/2013 --  

-- --  

-- Ce package définit les deux boîtes de dialogue du jeu :  

-- --Loose_Dialog qui s'ouvre si vous cliquez sur une mine et  

-- --min_Dialog qui s'ouvre si vous avez découvert toutes les  

-- --mines. Il fournit également les procédures nécessaires à  

-- --initialisation --
-----  

WITH Glib.Convert ; USE Glib.Convert ;
WITH Gnat.Strings ; USE Gnat.Strings ;
WITH Gtk.Image ; USE Gtk.Image ;
WITH Gtk.Button ; USE Gtk.Button ;
WITH Gtk.Radio_Button ; USE Gtk.Radio_Button ;
WITH Gtk.Label ; USE Gtk.Label ;
WITH Gtk.Notebook ; USE Gtk.Notebook ;
WITH Gtk.Box ; USE Gtk.Box ;
WITH P_Option ; USE P_Option ;
WITH P_Game.Methods ; USE P_Game.Methods ;  

PACKAGE BODY P_Dialog IS
  PROCEDURE Init_Loose_Dialog(Parent : Gtk.Window) IS
  BEGIN
    Gtk_New(Loose_Dialog,
      parent,
      Mode_Warn,
      Message_Warning,
      Buttons_Ok,
      Locale.To_Utf8("Vous avez sauté sur une mine !"));
    Loose_dialog.set_title("Perdu") ;
  END Init_Loose_Dialog ;  

  PROCEDURE Init_Win_Dialog(Parent : Gtk.Window) IS
  BEGIN
    Gtk_New(Win_Dialog,
      parent,
      Mode_Warn,
      Message_Warning,
      Buttons_Ok,
      Locale.To_Utf8("Vous avez trouvé toutes les mines !"));
    Win_dialog.set_title("Victoire") ;
  END Init_Win_Dialog ;  

  PROCEDURE Init_About_Dialog IS
    Liste_Auteurs : Gnat.Strings.String_List(1..1) ;
    Logo : GTK.Image ;
  BEGIN
    Gtk_New(Logo, "sea-mine.png") ;
    Gtk_New(About_Dialog) ;
    Liste_auteurs(1) := new_string'"Kajig"' ;
    About_Dialog.Set_Authors(Liste_Auteurs) ;
    About_Dialog.Set_Logo(Logo.Get) ;
    About_Dialog.Set_Program_Name(Locale_To_Utf8("Démineur Ada")) ;
    About_Dialog.Set_Version("2.0") ;
    About_Dialog.Set_Comments(locale_to_utf8("Vous trouverez le
code source
  & l'de ce programme sur le Site du Zéro "
  & à l'adresse suivante :
  & "http://www.v3.siteduzero.com/tutoriel-3-547729-1-
apprendre-a-programmer-avec-ada.html")) ;
  END Init_About_Dialog ;  

  PROCEDURE Init_Option_Dialog(Game : T_Game) IS
    Btn_Cancel : GTK.Button ; PRAGMA Unreferenced(Btn_Cancel)
    ;  

    Btn_OK : GTK.Button ; PRAGMA Unreferenced(Btn_OK) ;
    Onglets : GTK.Notebook ;
    Box1,box3,box4 : Gtk_VBox ;
    Box2 : Gtk_Hbox ;
    Page1 : Gtk_Label ;
    Page2 : Gtk_Label ;
    Btm_Facile : GTK_Radio_Button ;
    Btm_Difficile : GTK_Radio_Button ;
    Btm_Moyen : GTK_Radio_Button ;
    Btm_Drapeaul : GTK_Radio_Button ;
    Btm_Drapeau2 : GTK_Radio_Button ;
    Btm_Mine1 : GTK_Radio_Button ;
    Btm_Mine2 : GTK_Radio_Button ;
  BEGIN
    --Création de la boîte de dialogue
    Gtk_New(Option_Dialog) ;
    Option_Dialog_Set_Title(Locale_To_Utf8("Préférences")) ;
    --Création et ajout de deux boutons d'action
    Btn_Cancel := Gtk_Button(Option_Dialog.Add_Button("Annuler",
      GTK_Response_Cancel)) ;
    Btn_OK := Gtk_Button(Option_Dialog.Add_Button("Valider",
      GTK_Response_OK)) ;
    --Création et organisation des diverses boîtes nécessaires
    Gtk_New_Vbox(Box1) ;
    Gtk_New_Vbox(Box2) ;
    Gtk_New_Vbox(Box3) ;
    Gtk_New_Vbox(Box4) ;
    Box2.Pack_Start(Box3) ;
    Box2.Pack_Start(Box4) ;
    --Création, organisation et connexion des boutons radio de
difficulté
    --Puis sélection de la difficulté en cours
    Gtk_New(Btn_Facile, NULL, Locale_To_Utf8("Débutant")) ;
    Gtk_New(Btn_Moyen, Btn_Facile,Locale_To_Utf8("Confirmé")) ;
    Gtk_New(Btn_Difficile,Btn_Moyen, Locale_To_Utf8("Expert")) ;
    Box1.Pack_Start(Btn_Facile) ;
    Box1.Pack_Start(Btn_Moyen) ;
    Box1.Pack_Start(Btn_Difficile) ;
    Connect(Btn_Facile, "clicked",Change_Difficulty'ACCESS,1) ;
    Connect(Btn_Moyen, "clicked",Change_Difficulty'ACCESS,2) ;
    Connect(Btn_Difficile,"clicked",Change_Difficulty'ACCESS,3) ;
    CASE Game.General.Option(1) IS
      WHEN 1 => Btm_Facile.Set_Active(True) ;
      WHEN 2 => Btm_Moyen.Set_Active(True) ;
      WHEN OTHERS => Btm_Difficile.Set_Active(True) ;
    END CASE ;
    --Création, organisation et connexion des boutons radio de
drapeau
    Gtk_New(Btn_Drapeaul, NULL, "Drapeaux bleus") ;
    Gtk_New(Btn_Drapeau2, Btm_Drapeaul, "Drapeaux rouges") ;
    Box3.Pack_Start(Btn_Drapeaul) ;
    Box3.Pack_Start(Btn_Drapeau2) ;
  END Init_Option_Dialog ;

```

```

    Connect(Btn_Drapeau1,
    "toggled",Change_Drapeau_Filename'ACCESS,1) ;
    Connect(Btn_Drapeau2,
    "toggled",Change_Drapeau_Filename'ACCESS,2) ;
    CASE Game.General_Option(2) IS
        WHEN 1 => Btn_Drapeau1.Set_Active(True) ;
        WHEN OTHERS => Btn_Drapeau2.Set_Active(True) ;
    END CASE ;
    --Création, organisation et connexion des boutons radio de
mine
    Gtk_New(Btn_Mine1, NULL, "Mines rouges") ;
    Gtk_New(Btn_Mine2, Btn_Mine1, "Mines grises") ;
    Box4.Pack_Start(Btn_Mine1) ;
    Box4.Pack_Start(Btn_Mine2) ;
    Connect(Btn_Mine1, "toggled", Change_Mine_Filename'ACCESS,1) ;
    Connect(Btn_Mine2, "toggled", Change_Mine_Filename'ACCESS,2) ;
    CASE Game.General_Option(3) IS
        WHEN 1 => Btn_Mine1.Set_Active(True) ;
        WHEN OTHERS => Btn_Mine2.Set_Active(True) ;
    END CASE ;
    --Création de la barre d'onglets et des étiquettes
   Gtk_New(Page1,Locale_To_Utf8("Difficulté")) ;
   Gtk_New(Page2,"Apparence") ;
   Gtk_New(Onglets) ;
    Onglets.Append_Page(Box1,Page1) ;
    Onglets.Append_Page(Box2,Page2) ;
    --Finalisation
    Pack_Start(Option_Dialog.Get_Content_Area, Onglets) ;
    Option_Dialog.Show_All ;
END Init_Option_Dialog ;

PROCEDURE Run_About_Dialog(Emetteur : ACCESS
Gtk_Widget_Record'Class) IS
    PRAGMA Unreferenced(Emetteur) ;
BEGIN
    Init_About_Dialog ;
    IF About_Dialog.Run = Gtk_Response_close
    THEN NULL ;
    END IF ;
    About_Dialog.Destroy ;
    Gtk_New(About_Dialog) ;
END Run_About_Dialog ;

PROCEDURE Run_Option_Dialog(Emetteur : ACCESS
Gtk_Widget_Record'Class ;
    Game : T_Game) IS
    PRAGMA Unreferenced(Emetteur) ;
BEGIN
    Init_Option_Dialog(Game) ;
    IF Option_Dialog.Run = Gtk_Response_ok
    THEN Game.General_Option := Current_Option ;
        Set_Difficulty(Game) ;
        Set_Drapeau_Filename(Game.General_Option) ;
        Set_Mine_Filename(Game.General_Option) ;
        Reinit_Game(Game) ;
    END IF ;
    Option_Dialog.Destroy ;
    Gtk_New(Option_Dialog) ;
END Run_Option_Dialog ;

PROCEDURE Change_Difficulty(Emetteur : ACCESS
Gtk_Widget_Record'Class ;
    Valeur : Integer) IS
    PRAGMA Unreferenced(Emetteur) ;
BEGIN
    Current_Option(1) := Valeur ;
END Change_Difficulty ;

PROCEDURE Change_Drapeau_Filename(Emetteur : ACCESS
Gtk_Widget_Record'Class ;
    Valeur : Integer) IS
    PRAGMA Unreferenced(Emetteur) ;
BEGIN
    Current_Option(2) := Valeur ;
END Change_Drapeau_Filename ;

PROCEDURE Change_Mine_Filename(Emetteur : ACCESS
Gtk_Widget_Record'Class ;
    Valeur : Integer) IS
    PRAGMA Unreferenced(Emetteur) ;
BEGIN
    Current_Option(3) := Valeur ;
END Change_Mine_Filename ;

END P_Dialog ;

```

La procédure principale

Nous voilà enfin rendus à la procédure principale. Vous remarquerez que celle-ci c'est très nettement simplifiée : la partie console a été supprimée, les paramètres concernant la taille de la grille ou le nombre de bombes sont définis automatiquement au démarrage du jeu de même que les boîtes de dialogue sont générées automatiquement par la méthode Init_Game().

Code : Ada - Demineur.adb

```

WITH Gtk.Main ;
WITH P_Tile ;
WITH P_Game ;
WITH P_Game.Methods ;
    USE Gtk.Main ;
    USE P_Tile ;
    USE P_Game ;
    USE P_Game.Methods ;

PROCEDURE Demineur IS
    Game : CONSTANT T_Game := NEW T_Game_Record ;
BEGIN
    Drapeaux_restants := game.Bombs ;
    Init ;
    Init_Game(Game) ;
    Main ;
END Demineur ;

```

Pistes d'amélioration :

- permettre d'avantage de personnalisation des graphismes;
- chronométrier les parties et proposer un menu Scores affichant les meilleurs résultats;
- permettre de personnaliser la taille des grilles (par exemple 50 × 50 cases avec 2 bombes)

Voilà, nous avons fait un tour d'horizon très large de ce que le langage Ada pouvait permettre. J'espère vous avoir donné goût à ce langage ou, tout au moins, vous avoir fourni les bases pour programmer correctement, que vous continuiez votre chemin avec Ada ou que vous partiez vers d'autres ceux : C++, Java, Python...

Il n'est pas mauvais d'ailleurs de se frotter à d'autres langages, à leurs spécificités ou à leurs ressemblances. «*La Terre est le berceau de l'humanité, mais elle ne peut pas vivre contenue dans un berceau*» disait Constantin Tsiolkovski, père de l'astronautique russe. Il en est de même en programmation, quel que soit votre langage favori, vous serez amenés à en utiliser d'autres. Il n'y a qu'un peu que vous connaîtrez celui qui est fait pour vous.

Ce long cours touche à sa fin. Après toutes ces heures de lecture, vous avez pu découvrir les rudiments de l'algorithme et du langage Ada (dans sa version 83), puis vous avez découvert les fonctionnalités offertes par les normes Ada 95 et 2005 en matière de programmation orientée objet. Enfin, vous avez pu découvrir une bibliothèque tierce et la programmation événementielle au travers de GTKAda. Que vous reste-t-il à apprendre ? Encore beaucoup à vrai dire. En programmation comme en toute chose, on n'a jamais fini d'apprendre. Mais voici quelques pistes :

- Tout d'abord, d'autres bibliothèques graphiques existent et ont été traitées sur ce site. Par exemple la SDL, traitée dans le cours de M@eo21 sur le C, dispose d'une version Ada. De même, il existe des interfaces de QT pour Ada appellé QT4Ada (QT est traité dans le cours sur le C++). Le web regorge de ressources, à vous de les découvrir.
- Ensuite, d'autres bibliothèques existent et permettent d'effectuer divers travaux. Ainsi la société Adacore fournit gratuitement la bibliothèque XML/Ada pour traiter des fichiers XML et cela tombe bien car le Site du Zéro propose quant

à lui un excellent tutoriel d'introduction au langage XML. Des bibliothèques permettent également de travailler sur les bases de données MySQL ou sur le web.

- Enfin, il est important que vous découvriez d'autres langages de programmation. Cela vous permettra de découvrir de nouvelles façons de penser, de travailler ... et le Site du Zéro regorge d'excellents tutuels sur divers langages comme C, C++, Python, Java, C#, OCaml ... Cela vous permettra aussi de vous rendre compte de tout ce que programmer en Ada vous aura appris en termes de rigueur ou de structuration de votre code.

J'espère que ce cours vous aura plu et qu'il vous aura donné les outils pour aborder ces nouveaux domaines de la programmation avec sérénité. Je remercie enfin le Site du Zéro pour les conseils et soutiens qu'il m'a aura apporté tout au long de la rédaction de ce tutoriel.