

4D Compiler

Manuel de référence
Windows®/Mac™ OS



4D Compiler

Manuel de référence

Copyright© 2000 4D SA/4D, Inc.
Tous droits réservés.

Les informations contenues dans ce manuel peuvent faire l'objet de modifications sans préavis et ne sauraient en aucune manière engager 4D SA. La fourniture du logiciel décrit dans ce manuel est régie par un octroi de licence dont les termes sont précisés par ailleurs dans la licence électronique figurant sur le support du Logiciel et de la Documentation y afférente. Le logiciel et sa Documentation ne peuvent être utilisés, copiés ou reproduits sur quelque support que ce soit et de quelque manière que ce soit, que conformément aux termes de cette licence.

Aucune partie de ce manuel ne peut être reproduite ou recopiée de quelque manière que ce soit, électronique ou mécanique, y compris par photocopie, enregistrement, archivage ou tout autre procédé de stockage, de traitement et de récupération d'informations, pour d'autres buts que l'usage personnel de l'acheteur, et ce exclusivement aux conditions contractuelles, sans la permission explicite de 4D SA.

4D, 4D Calc, 4D Draw, 4D Write, 4D Insider, 4ème Dimension, 4D Server, 4D Compiler ainsi que les logos 4e Dimension et 4D sont des marques enregistrées de 4D SA.

Windows, Windows NT, Win 32s et Microsoft sont des marques enregistrées de Microsoft Corporation.

Apple, Macintosh, Power Macintosh, LaserWriter, ImageWriter, QuickTime sont des marques enregistrées ou des noms commerciaux de Apple Computer, Inc.

Mac2Win Software Copyright © 1990-2000 est un produit de Altura Software, Inc.

ACROBAT © Copyright 1987-2000 , Secret Commercial Adobe Systems Inc. Tous droits réservés. ACROBAT est une marque enregistrée d'Adobe Systems Inc.

Tous les autres noms de produits ou appellations sont des marques déposées ou des noms commerciaux appartenant à leurs propriétaires respectifs

Sommaire

Chapitre 1	Introduction.9
	Avant-propos	9
	4D Compiler : un principe simple	9
	Conseils d'approche.	9
	A propos de ce manuel	10
	Windows®/Mac™OS	10
	Navigation hypertexte.	10
	Contenu du manuel.	11
	Qu'est-ce qu'un compilateur?	12
	Programme interprété, programme compilé	12
	Objets compilés	13
	4D Compiler (MacOS ou Windows)	13
	Code généré	13
	Pourquoi compiler vos bases ?.	14
	Vitesse	14
	Vérification de votre code.	15
	Protection de vos applications	16
	Compilation en mode Fat binary	16
	Exercice de manipulation	17
	Ouverture de l'exemple Démonstration sous 4 ^e Dimension.	17
	Compilation de l'exemple.	18
	Utilisation de la base compilée	20
	A vous de jouer	21
 Chapitre 2	 Notions de base.23
	Principe de base : un objet, un nom, une seule identité	23
	Typage des variables.	24
	Rappel : les types des variables	24
	Création de la table des symboles.	25
	Le typage des variables par 4D Compiler	26

Directives de compilation	27
Les commandes	27
Quand utiliser des directives de compilation ?	28
Où placer vos directives de compilation ?	33
Une syntaxe particulière : C_ALPHA.	35
Une liberté permise par 4D Compiler	37
Conclusion	37

Chapitre 3

Guide du typage. 39

Conflits sur les variables simples.	39
Conflit entre deux utilisations.	39
Conflit entre une utilisation et une directive de compilation	39
Conflit par retypage implicite	40
Conflit entre deux directives de compilation	41
Note sur les variables locales.	42
Conflits sur les variables tableau.	42
Changer le type des éléments d'un tableau	43
Changer le nombre de dimensions d'un tableau	43
Cas des tableaux de chaînes fixes	44
Retypes implicites	44
Tableaux locaux	44
Typage des variables dessinées dans les formulaires.	45
Variables considérées par défaut comme des Numériques	45
Variable Graphe	46
Variable Zone externe	46
Variables considérées par défaut comme du Texte	47
Questions de type autour des pointeurs	48
Plug-ins.	49
Généralités	49
Compilation multi-plate-forme	50
Routines recevant des paramètres implicites	54
Variables créées par des commandes de plug-ins	55
Composants 4D	55
Manipulation des locales \$0...\$N et passation des paramètres	56
Utiliser les pointeurs pour éviter les retypes	56
Indirections sur les paramètres	58
Variables réservées et constantes.	60
Variables système	60
Variables des états semi-automatiques.	60
Constantes 4D	60

Chapitre 4	Précisions de syntaxe	61
	Chaînes de caractères	61
	Communications	62
	Définition structure	63
	Documents système	63
	Fonctions mathématiques	64
	Interruptions.	64
	Tableaux	66
	Utilisation des pointeurs	
	dans les commandes s'appliquant aux tableaux	68
	Tableaux locaux	68
	Langage	69
	Variables	71
	Précisions concernant l'utilisation de pointeurs	73
	Constantes	74
 Chapitre 5	 Conseils d'optimisation	 75
	Remarque préliminaire : les commentaires	76
	Optimisation par les directives de compilation	76
	Numériques	76
	Chaînes de caractères	78
	Remarques diverses	79
	Tableaux à deux dimensions	79
	Champs	79
	Pointeurs.	80
	Variables locales	80
 Chapitre 6	 Fenêtres des options	 81
	Remarques préliminaires	81
	Les menus de 4D Compiler	81
	Nouveau...	82
	Ouvrir...	83
	Recompiler.	85
	Fermer	85
	Enregistrer	85
	Enregistrer sous...	85
	Version précédente	85
	Projet par défaut.	85
	Quitter	86
	Fonctionnalités d'interface	86
	Drag and drop (glisser-déposer)	86

Les fenêtres d'options pour la compilation	87
Première fenêtre d'options	87
Deuxième fenêtre d'options	96
Remarques générales sur les options	101
Lancement d'une compilation	101
Le processus de compilation	102
Remarque préliminaire	102
Copie de la base	103
Typage des variables	103
Compilation	105
Génération d'un exécutable	107
Utilisation d'une base compilée	107

Chapitre 7

Les aides à la compilation 109

La table des symboles	109
La liste des variables process et interprocess	110
La liste des variables locales	111
La liste complète des méthodes	111
Le fichier d'erreurs	112
Les types de messages	112
L'exploitation du fichier d'erreurs	115
Le fichier de typage	119
Le contrôle d'exécution	120
Définition et fonction du contrôle d'exécution	120
Quand et comment tirer parti du contrôle d'exécution ?	121
Pour mémoire : en cas d'anomalies	121

Chapitre 8

Messages 123

Les warnings	123
Les warning plus	124
Les erreurs	125
Typage	125
Syntaxe	127
Paramètres	129
Opérateurs	130
Plug-ins	130
Erreurs générales	131
Les messages du contrôle d'exécution	132
Les messages d'alerte	134

Annexe A	Personnaliser son application	135
	Personnaliser son application sous Mac TM OS	135
	Changer la signature de l'application	135
	Changer l'icône de l'application	136
	Changer les icônes des fichiers annexes	136
	Personnaliser son application sous Windows	137
	Changer l'icône de l'application	137
	Changer le nom de la fenêtre de l'application	138
 Annexe B	 Personnaliser 4D Compiler	 139
	Traduction	140
 Index		 141

1

Introduction

Avant-propos

4D Compiler : un principe simple

4D Compiler est un compilateur pour 4^e Dimension et 4D Server.

Le terme est savant, mais la réalité simple. L'emploi de ce compilateur est transparent, que vous soyez un utilisateur profane ou un spécialiste. Vous accédez à la base que vous voulez compiler par une boîte de dialogue standard d'ouverture de fichiers. Ensuite, le compilateur s'occupe de tout : il duplique la base, traduit les méthodes de la copie en langage machine et génère, éventuellement, le fichier des erreurs.

L'accélération en exploitation d'une base compilée est très significative, le facteur d'accélération se situant entre 3 et 1000, et parfois davantage, selon les opérations.

4D Compiler est un compilateur au sens strict du terme, générant du vrai code assembleur, adapté au microprocesseur de la machine.

Conseils d'approche

La facilité d'interface de 4D Compiler et la clarté des principes de base qu'il convient de respecter ne signifient pas que vous allez utiliser en mode compilé une base que vous avez développée, dans les cinq minutes qui suivent l'acquisition du produit, même si vous êtes un développeur 4^e Dimension chevronné.

4D Compiler s'occupera de tout effectivement, et en particulier de vous signaler vos erreurs de code, de les resituer dans leur contexte, mais il ne pourra pas les corriger pour vous...

Le nombre d'erreurs de compilation lors de vos premières utilisations de 4D Compiler, sera certainement spectaculaire. Ne vous affolez pas pour autant. Vous découvrirez très rapidement qu'elles ont souvent la même origine, le non respect de quelques conventions de base.

4D Compiler vous fournira toujours un diagnostic précis vous permettant de les corriger.

L'utilisation optimale de 4D Compiler suppose que vous considériez les points suivants :

- une base qui fonctionne mal sous 4^e Dimension ne fonctionnera pas mieux compilée. Il y a des chances, d'ailleurs, pour qu'elle ne soit pas compilable : 4D Compiler notera des erreurs qui l'empêcheront d'aller jusqu'au bout du processus de compilation.
- une base qui fonctionne parfaitement sous 4^e Dimension ne sera pas toujours compilable du premier coup, surtout au début.

En effet, 4^e Dimension est très souple en matière de cohérence sémantique et syntaxique. Si 4D Compiler est d'une exceptionnelle flexibilité, rare pour un compilateur, il ne peut être aussi tolérant que 4^e Dimension. La raison en est, non une limite de 4D Compiler, mais une différence essentielle entre le mode interprété et le mode compilé. Nous expliquons cette différence dans la suite de cette introduction.


A propos de ce manuel

Windows®/Mac™ OS Ce manuel s'adresse indifféremment aux utilisateurs des versions Windows et MacOS (Power Macintosh) de 4D Compiler. Les explications s'appliquent aux deux plates-formes. Toute différence de fonctionnement entre les versions MacOS et Windows de 4D Compiler est toutefois signalée au cours du texte.

Les copies d'écran proviennent principalement de l'environnement Windows (Windows 95). La version MacOS d'un écran n'est présentée que lorsqu'elle comporte des différences majeures avec Windows.

Navigation hypertexte

Si vous consultez ce manuel sous sa forme électronique (Acrobat), vous pouvez tirer profit des liens hypertexte qu'il contient. Dans les chapitres de ce manuel, chaque mot comportant un lien hypertexte apparaît en bleu, par exemple : "reportez-vous à la [page 12](#)" (ce principe ne s'applique pas aux parties "Sommaire" et "Index", dans lesquelles toutes les entrées comporte un lien).

Lorsque vous cliquez sur un lien hypertexte, vous vous déplacez instantanément sur une page comportant des informations supplémentaires. Pour retourner à la page de départ, il vous suffit de cliquer sur le bouton **Page précédente** d'Acrobat 

Vous pouvez également vous déplacer en cliquant sur les repères dans la table située à gauche de la fenêtre affichant les pages du manuel.

Contenu du manuel

Poursuivez la lecture de cette introduction. Elle vous permet :

- de mieux comprendre la différence entre un programme compilé et un programme interprété et l'intérêt de compiler ses bases.
- d'avoir un premier contact avec le compilateur par la compilation d'une base simple, appelée DEMO.4DB si vous travaillez sous Windows et Démonstration sous MacOS.
- Les chapitres 2 à 5 sont consacrés à l'explication des notions de base qu'il convient de connaître pour tirer parti des possibilités du compilateur :
 - **“Notions de base”, page 23**
Ce chapitre est intégralement consacré à l'explication du principe le plus important pour compiler une base : donner au compilateur les moyens d'identifier de façon sûre ce qu'il doit compiler.
 - **“Guide du typage”, page 39**
Ce chapitre pratique recense les causes les plus fréquentes d'erreurs.
 - **“Précisions de syntaxe”, page 61**
Ce chapitre apporte des précisions sur le comportement de certaines routines de 4^e Dimension lorsque vous concevez une application qui doit être compilée.
 - **“Conseils d'optimisation”, page 75**
Ce chapitre propose des exemples simples d'amélioration de code.
- Les chapitres 6 à 8 sont consacrés à l'explication de l'ensemble des fonctionnalités et options proposées par le compilateur :
 - **“Fenêtres des options”, page 81**
Ce chapitre décrit les options de compilation de 4D Compiler et leur fonctionnement.
 - **“Les aides à la compilation”, page 109**
Ce chapitre décrit les différents outils d'aide à l'analyse, à la correction et à l'exécution mis à votre disposition pour un “débogage” rapide et complet de vos bases.

- [“Messages”, page 123](#)

Ce chapitre dresse la liste des messages délivrés par 4D Compiler au cours de son travail.

- Enfin, des annexes vous permettent d'utiliser au mieux le compilateur :

- [annexe A, “Personnaliser son application”, page 135](#)

- [annexe B, “Personnaliser 4D Compiler”, page 139](#)

Qu'est-ce qu'un compilateur?

Programme interprété, programme compilé

Un ordinateur est une machine à laquelle on ne peut transmettre des ordres que par des “0” et des “1”. Le cœur de cette machine, le microprocesseur, n'est pas capable de comprendre un autre langage. Ce langage binaire est appelé *langage machine*.

Lorsque l'on écrit un programme, dans n'importe quel langage informatique (C, C++, Pascal, BASIC, 4^e Dimension...), les instructions sont traduites en langage machine, afin d'être comprises par le microprocesseur de l'ordinateur.

Deux cas de figure peuvent alors se présenter :

- si les instructions composant le programme sont traduites au fur et à mesure de leur exécution, on dit alors que le programme est *interprété*.
- si les instructions sont traduites en bloc avant le lancement de l'exécution du programme, on dit alors que le programme est *compilé*.

Un programme interprété

Pour un programme interprété, l'exécution d'une instruction se décompose de la façon suivante :

- lecture de l'instruction dans le langage du programme,
- traduction de l'instruction en langage machine,
- exécution de l'instruction.

Ce cycle s'exécute pour chacune des instructions composant un programme. Le programme qui se charge de l'exécution d'un tel cycle pour chaque instruction est appelé *interpréteur*. Jusqu'à ce jour, les méthodes de vos bases 4^e Dimension étaient interprétées.

Un programme compilé	<p>Un programme compilé est traduit dans sa globalité avant le début de l'exécution. On obtient alors une série d'instructions en langage machine. Cette série peut être conservée et utilisée plusieurs fois.</p> <p>Ainsi, pour un programme donné, la phase de traduction n'est effectuée qu'une seule fois, quel que soit le nombre d'utilisations futures du programme.</p> <p>Cette phase est totalement indépendante de toute utilisation du programme. L'application chargée de la traduction est appelée <i>compilateur</i>.</p>
Objets compilés	<p>4D Compiler compile dans votre base, développée sous 4^e Dimension, les méthodes base, les méthodes projet, les méthodes table, les méthodes formulaire et les méthodes objet. Si vous n'avez aucun de ces éléments dans une application, 4D Compiler n'a rien à compiler.</p> <p>Lorsque vous avez mené à bien cette compilation, l'utilisation de la base compilée est identique à celle de la base originale.</p>
4D Compiler (MacOS ou Windows)	<p>4D Compiler version MacOS ou Windows permet de générer du langage machine pour les plates-formes MacOS et Windows.</p> <p>De plus, 4D Compiler version MacOS permet de générer, à partir de votre base compilée et de 4D Engine version MacOS, une application autonome exécutable sous MacOS.</p> <p>4D Compiler version Windows permet de générer, à partir de votre base compilée et de 4D Engine version Windows, une application exécutable sous Windows.</p>
Code généré	<p>4D Compiler versions MacOS et Windows génèrent :</p> <ul style="list-style-type: none">■ du code PowerPC ;■ du code 80386, 80486 et Pentium.

Pourquoi compiler vos bases ?

Le premier bénéfice de la compilation est évidemment un gain de temps conséquent lors de l'exécution des instructions. Il existe deux autres avantages directement liés à la compilation :

- la vérification systématique de votre code,
- la protection de vos bases.

Vitesse

Le gain de performance tient à deux raisons majeures :

- la traduction directe et définitive du code,
- l'accès direct aux adresses des variables et des méthodes.

Traduction directe et définitive du code

Le code des méthodes écrites sous 4^e Dimension, sera, avec le compilateur, traduit une fois pour toutes. Le temps passé à traduire chacune des instructions, au fur et à mesure qu'elles apparaissent, sera économisé lors de l'utilisation de votre base. Voyons ici deux exemples où ce gain de temps sera appréciable.

Imaginons une boucle contenant une séquence d'instructions devant être exécutée 50 fois :

Boucle (\$i;1;50)

 `Séquence d'instructions

Fin de boucle

Chaque instruction de la séquence est traduite 50 fois dans une base interprétée. Avec le compilateur, la phase de traduction de chaque instruction de la séquence est supprimée. On économise ainsi, pour chaque instruction de la séquence, 50 traductions.

Imaginons maintenant un autre cas : votre base 4^e Dimension comporte une méthode base "Sur ouverture". Cette méthode est exécutée chaque fois que vous entrez dans votre base. Après la compilation, chaque fois que vous entrerez dans votre base, vous économiserez le temps de traduction de la méthode base "Sur ouverture". Il ne s'agit bien entendu ici que de deux exemples. En fait, cette accélération est sensible pour toutes les instructions de 4^e Dimension.

Accès direct aux
adresses des variables
et des méthodes

Dans une base non compilée, l'accès aux variables s'effectue par l'intermédiaire d'un nom. Il faut donc accéder au nom pour accéder à la valeur de la variable.

Dans le code compilé, le compilateur attribue à chaque variable une adresse, écrit directement l'adresse de la variable dans le code et va directement à cette adresse, lorsqu'un accès à la variable est nécessaire.

Note Il va de soi que les opérations nécessitant des accès à vos disques seront peu accélérées car leur vitesse est limitée par la vitesse physique de transmission de votre ordinateur vers ses périphériques (lecteur de disquettes ou disque dur).

Note Les commentaires ne sont pas traduits et n'apparaissent pas dans le code compilé. Ces lignes n'interviennent en rien dans les temps d'exécution en mode compilé.

Vous trouverez ci-après un tableau figurant des temps d'exécution en mode interprété et en mode compilé. Les tests ont été effectués sur un PC Pentium 90 et sur un Power Macintosh 7100/66. Nous vous présentons ici les deux cas extrêmes : une boucle simple et une boucle ne faisant pratiquement que des accès disque.

Ordinateur	Séquence	Interprété	Compilé	Facteur d'accélération
Power Macintosh 7100/66	<code>\$i:=0</code> Tant que (<code>\$i#50000</code>) <code>\$i:=\$i+1</code> Fin tant que	2 mn	0 s	–
Pentium 90	<code>\$i:=0</code> Tant que (<code>\$i#20000</code>) <code>\$i:=\$i+1</code> CREER ENREGISTREMENT Champ1:= ... STOCKER ENREGISTREMENT Fin tant que	1 mn 30 s	41 s	2,1

Vérification
de votre code

Le compilateur fonctionne aussi comme analyseur syntaxique de vos bases. Il procède à un examen systématique de votre code et relève vos éventuelles négligences alors que 4^e Dimension ne le fait que lorsque la méthode est exécutée.

Supposez que l'une de vos méthodes contienne une série de tests et des séquences d'instructions à exécuter. Il est improbable que vous testiez absolument tous les cas de figure si le nombre de tests est, par exemple, supérieur à 100. Il n'est alors pas impossible que se révèle à l'utilisation une erreur de syntaxe contenue dans un cas non testé.

Ce phénomène ne peut pas se produire lorsque vous utilisez une base compilée. En effet, lors de la compilation, la base entière est parcourue et la syntaxe de chaque instruction est vérifiée. S'il existe une anomalie, le compilateur la détecte et vous demande d'y remédier.

Protection de vos applications

La base compilée est une copie de votre base originale, à ceci près qu'elle verrouille l'accès au mode Structure. Dans une base compilée, la commande **Structure** du menu **Mode** est donc désactivée.



Les avantages sont les suivants :

- la structure de la base ne peut être modifiée volontairement ou involontairement,
- vos méthodes sont désormais protégées.

Compilation en mode *Fat binary*

4D Compiler vous permet de sélectionner un ou deux générateurs de code en même temps : 386/486 et PowerPC par exemple.

Dans ce cas, 4D Compiler générera du "fat binary", c'est-à-dire, dans la même base, à la fois le code PowerPC et le code 80x86 de votre choix. Bien entendu, lors de l'exploitation de votre base, c'est le code adapté au processeur de la machine de l'utilisateur qui est exécuté.

Si la base compilée doit être utilisée avec 4D Server, le code adéquat sera chargé et exécuté sur chacun des postes clients : PowerPC sur Power Macintosh, et 80x86 sur PC.

Il est possible de désactiver les deux icônes de choix du code. Dans ce cas, 4D Compiler effectuera le chemin de compilation, sans générer la base compilée, mais pourra créer les fichiers d'aide (fichier d'erreurs, table des symboles...) que vous lui aurez demandés.

Exercice de manipulation

Ouverture de l'exemple Démonstration sous 4^e Dimension

L'exemple qui vous est fourni avec 4D Compiler est une petite base 4^e Dimension. Il a pour but de vous montrer comment compiler une base puis l'utiliser. Cet exemple n'a pas pour fonction de vous décrire toutes les fonctionnalités du compilateur, mais simplement de vous mettre en situation.

1 Lancez votre logiciel 4^e Dimension.

2 Ouvrez la base de démonstration.

Vous accédez à l'application en mode Menus créés. Déroulez le menu **Fichier** et choisissez **Démonstration**. Vous disposez alors d'un tableau de valeurs sur lesquelles vous allez faire une simulation qui obligera à un recalcul de tous les éléments du tableau.

Menus créés					
	1er trimestre	2e trimestre	3e trimestre	4e trimestre	
afrique du sud	30880	19261	31109	1512	0
albanie	19375	5263	23665	8320	0
algérie	9014	14038	4862	22643	0
allemagne	1101	26472	15445	4748	0
r. tchèque	12457	25487	1245	1212	0
angleterre	3102	25568	8110	4108	0
arabie saoudite	20535	24031	319	21165	0
argentine	16977	7688	29339	7567	0
australie	21557	32711	20609	25732	0
autriche	9246	11085	14151	14615	0
bahrein	32311	11943	19458	17081	0
bangladesh	11983	8870	8154	5955	0
belgique	7949	25895	19684	15451	0
bénin	30778	19081	31941	28641	0
birmanie	10721	13326	24223	3840	0
brésil	22604	21232	10624	19529	0
bulgarie	1043	30975	30455	18375	0
cambodge	23116	24655	1059	31841	0
cameroun	2231	25251	13099	3165	0
canada	9525	31513	10258	26778	0
	25579	11146	28187	27291	0

Ventes 1998

Simulation

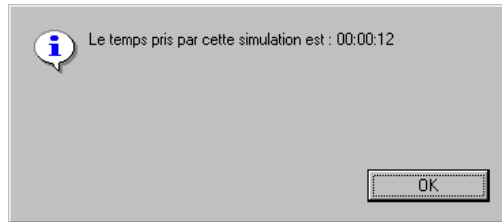
Fin

3 Cliquez sur le bouton Simulation.

Une boîte de dialogue vous demande d'indiquer le pourcentage de modification de ces valeurs.

4 Saisissez une valeur quelconque.

Lorsque l'opération est terminée, notez à l'écran le temps qu'elle a pris.



Note Cette durée varie selon la machine que vous utilisez.

5 Quittez la base d'exemple.

Nous allons maintenant compiler cette base.

Compilation de l'exemple



1 Lancez le logiciel 4D Compiler en double-cliquant sur l'icône du programme.

2 Déroulez le menu Fichier et choisissez la commande Nouveau.

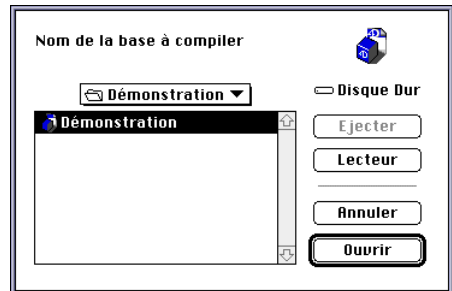
Une boîte de dialogue standard d'ouverture de documents apparaît : 4D Compiler vous demande de localiser la base à compiler.

3 Sélectionnez la structure de la base dans le dossier "Demo" sous Windows ou "Démonstration" sous MacOS :

Windows



MacOS



4 Cliquez sur le bouton Ouvrir.

4D Compiler affiche alors la fenêtre de définition des options.

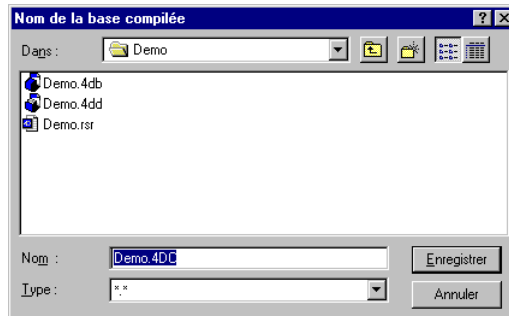
5 Cliquez sur l'icône "Nom de la base compilée" :

Bouton permettant de
baptiser la base compilée



6 Une boîte de dialogue de création de documents s'affiche.

Windows



MacOS



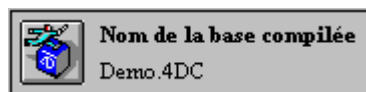
Par défaut, le nom de la base compilée est :

- sous Windows, Demo.4DC (c'est-à-dire *NomDeLaBase.4DC*),
- sous MacOS, Démonstration.comp (c'est-à-dire *NomDeLaBase.comp*).

A votre convenance, vous pouvez laisser ou non le nom donné par défaut.

7 Cliquez sur le bouton Enregistrer.

Vous revenez à la fenêtre des options : le nom de la base compilée est inscrit à droite de l'icône.



Pour cet exercice, ne vous occupez pas des autres icônes de la fenêtre d'options ; leur fonction sera décrite ultérieurement.

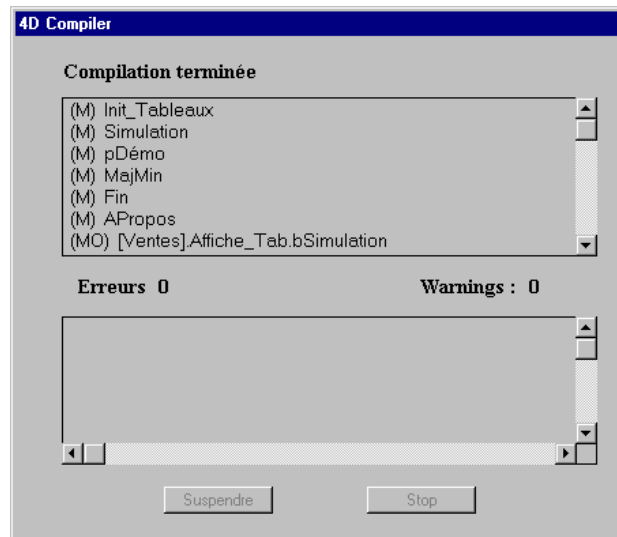
8 Cliquez sur le bouton OK.

Une boîte de dialogue vous propose d'enregistrer un *projet*, un projet étant la sauvegarde du choix des options. Vous n'en n'avez pas besoin dans le cadre de cet exercice.

9 Cliquez sur le bouton Non.

Dès que vous avez cliqué sur **Non**, vous entrez dans le processus de compilation. La compilation se déroule alors sans votre intervention, passant en revue toutes les méthodes de la base. Les différentes étapes vous seront décrites ultérieurement.

Le compilateur vous signale par le message **Compilation terminée** que son travail est fini.



10 Demandez Quitter dans le menu Fichier.

Utilisation de la base compilée

Pour utiliser une base compilée, vous devez disposer d'une version 6 ou ultérieure de 4^e Dimension.

1 Lancez 4^e Dimension et ouvrez la base Démonstration.comp ou Demo.4DC.

L'utilisation de la base compilée n'est en rien différente de l'utilisation d'une base non compilée.

Refaites les opérations décrites plus haut et comparez les temps d'exécution.

Note Sous Windows, au premier lancement d'une base compilée, 4^e Dimension crée un fichier nommé NomdeLaBase.CMP. Ce fichier est nécessaire pour l'exécution d'une base compilée.

A vous de jouer

Dès que vous en aurez terminé avec ce bref exemple, vous serez impatient d'essayer une de vos bases et les conseils les plus insistants ne vous en dissuaderont pas.

La seule recommandation que nous puissions vous faire, est que vous vous munissiez d'un peu d'humour.

Il est improbable que votre base, aussi belle soit-elle, soit compilable du premier coup.

Vous aurez peut-être 100 ou 2 000 erreurs. Ce nombre, cependant, ne reflétera en rien ni votre talent, ni votre capacité ultérieure à utiliser 4D Compiler.

Alors n'accusez personne :

- ni 4D Compiler,
- ni 4^e Dimension,
- ni vous-même ou ceux qui ont créé la base pour vous.

La solution la meilleure est de prendre connaissance des règles qui vous éviteront les multiples erreurs.

2

Notions de base

Principe de base : un objet, un nom, une seule identité

Lorsque vous écrivez une application avec 4^e Dimension, vous baptisez comme vous le voulez vos méthodes et vos variables.

Vous conservez naturellement la même liberté avec le compilateur, à ceci près qu'il exige de vous plus d'imagination encore :

- Vous ne devez jamais donner deux fois le même nom à vos méthodes ou à vos variables de nature et de fonction différentes.
- Vous ne devez pas avoir une méthode qui aurait le même nom qu'une variable.
- Vous ne devez pas avoir une routine externe qui porterait le même nom qu'une méthode ou une variable.

Comme nous le verrons au cours de cette section — et vous le rappellerons constamment dans ce manuel — il est indispensable que le compilateur puisse identifier sans ambiguïté les objets qui lui sont proposés.

Les objets sur lesquels vous avez le plus de risques de générer des ambiguïtés gênant le compilateur dans son travail d'identification sont les variables. C'est pourquoi cette section leur est consacrée.

Ces ambiguïtés tiennent à deux causes majeures :

- attribution d'un même nom à deux variables différentes,
- attribution de types différents à une même variable.

Typage des variables

4^e Dimension utilise trois catégories de variables :

- les variables locales,
- les variables process,
- les variables interprocess.

Note Si vous ne maîtrisez pas cette notion de catégories de variables, nous vous recommandons de vous reporter au manuel *Langage* de 4^e Dimension avant de poursuivre la lecture du présent manuel.

Les variables process et les variables interprocess sont structurellement de même nature pour 4D Compiler. Dans le guide du typage, en particulier, les mêmes remarques s'appliquent à l'une ou à l'autre de ces catégories de variables.

Rappel : les types des variables

Toutes les variables ont un type.

Comme vous avez pu le voir dans la documentation de 4^e Dimension, vous disposez de 12 types pour les variables simples :

- Booléen
- Alphanumérique (ou Chaîne fixe)
- Date
- Entier
- Entier long
- Graphe
- Heure
- Image
- Numérique (ou Réel)
- Pointeur
- Texte
- BLOB

Pour les variables de type Tableau, vous disposez des 9 types suivants :

- Tableau Booléen
- Tableau Alpha (ou Chaîne fixe)
- Tableau Date
- Tableau Entier
- Tableau Entier long
- Tableau Image
- Tableau Numérique (ou Réel)
- Tableau Pointeur
- Tableau Texte

Création de la table des symboles

Lorsque vous travaillez avec 4^e Dimension, une variable de même nom peut avoir plusieurs types. Cette tolérance se justifie parfaitement puisque vous êtes en mode interprété. En effet, à chaque ligne de code, 4^e Dimension interprète l'instruction et comprend le contexte.

Lorsque vous travaillez en mode compilé, vous êtes dans une situation différente. Alors que l'interprétation agit ligne par ligne, la compilation s'intéresse à une base dans sa globalité.

La manière d'opérer de 4D Compiler est la suivante :

- Le programme explore systématiquement les objets qui lui sont proposés par 4^e Dimension.
Ces objets sont les méthodes base, les méthodes projet, les méthodes formulaire, les méthodes table (triggers) et les méthodes objet.
- Le programme peigne ces objets pour retrouver le type de chacune des variables utilisées dans la base et génère la table des variables et des méthodes.
- Une fois qu'il a retrouvé le type de toutes les variables, le compilateur traduit la base. Encore faut-il qu'il puisse identifier un type d'une manière univoque pour chacune des variables.

Si le compilateur trouve un même nom de variable avec deux types différents, il n'a aucune raison de privilégier l'un par rapport à l'autre. En d'autres termes, avant de pouvoir ranger un objet, de lui donner une adresse mémoire, le compilateur a besoin de connaître l'identité

exacte de cet objet, c'est-à-dire son nom et son type, le type permettant au compilateur de déduire sa taille.

Ainsi, pour chaque application compilée, le compilateur crée un plan, contenant, pour chaque variable, son nom (ou identificateur), son emplacement (ou adresse mémoire) et la taille qu'elle occupe (représentée par son type). Ce "plan" s'appelle *table des symboles*.

Le typage des variables par 4D Compiler

4D Compiler, comme tout compilateur, doit donc respecter les critères d'identification des variables.

Vous avez alors deux possibilités :

- Si vos variables ne sont pas typées, 4D Compiler s'en occupera automatiquement pour vous. Toutes les fois que c'est possible, dans la mesure où il n'y a pas d'ambiguïtés, 4D Compiler déduit automatiquement pour vous le type des variables utilisées. Si, par exemple, vous écrivez :

V1 := Vrai

le compilateur pourra en déduire que la variable V1 est de type Booléen.

De même, si vous écrivez :

V2:= "Ceci est une phrase exemple"

le compilateur en déduira que V2 est une variable de type Texte.

Le programme peut également déduire le type des variables dans des cas moins faciles :

V3:= V1

'V3 est du même type que V1

V4:= 2*V2

'V4 est du même type que V2

4D Compiler déduit également le type de vos variables d'après les appels aux commandes 4^e Dimension et à vos propres méthodes.

Si vous passez à une méthode un paramètre de type Booléen et un paramètre de type Date, 4D Compiler donnera aux variables locales \$1 et \$2 de la méthode appelée le type Booléen et le type Date.

Lors de ces déductions, sauf indication contraire dans le projet de compilation, le compilateur ne donne pas à vos variables des types limitatifs : Entier, Entier long ou Alphanumérique. Par défaut, 4D Compiler donne toujours le type le plus large possible à vos variables. Si vous écrivez :

```
LeNombre :=4
```

le compilateur donnera à la variable LeNombre le type Réel, même si en toute rigueur la valeur 4 est entière. En d'autres termes, le compilateur n'exclut pas que dans d'autres occurrences de la variable, la valeur puisse être 4,5.

Evidemment, si vous ne voulez pas de cette interprétation générale, vous pouvez préciser vos choix : c'est l'objet de ce qu'on appelle les *directives de compilation*. Pour en savoir plus sur ce sujet reportez-vous au [chapitre "Guide du typage", page 39](#).

- Si vous avez déjà typé vos variables et pensez que votre typage est cohérent et complet, vous pouvez explicitement demander à ce que 4D Compiler ne refasse pas ce travail.
Pour le cas où votre typage, contrairement à ce que vous croyiez, n'aurait pas été suffisant, vous aurez les erreurs de compilation vous invitant à apporter les modifications nécessaires.

Directives de compilation

Les commandes

Vous avez déjà eu un aperçu des directives de compilation dans le manuel *Langage* de 4^e Dimension.

Il s'agit des commandes suivantes :

```
C_ALPHA({Méthode;}Longueur;Var1 {;Var2;...;VarN})
```

```
C_BOOLEEN({Méthode;}Var1 {;Var2;...;VarN})
```

```
C_DATE({Méthode;}Var1 {;Var2;...;VarN})
```

```
C_ENTIER({Méthode;}Var1 {;Var2;...;VarN})
```

```
C_HEURE({Méthode;}Var1 {;Var2;...;VarN})
```

```
C_IMAGE({Méthode;}Var1 {;Var2;...;VarN})
```

C_ENTIER LONG({Méthode;}Var1 {;Var2;...;VarN})

C_REEL({Méthode;}Var1 {;Var2;...;VarN})

C_POINTEUR({Méthode;}Var1 {;Var2;...;VarN})

C_TEXTE({Méthode;}Var1 {;Var2;...;VarN})

C_GRAPHE({Méthode;}Var1 {;Var2;...;VarN})

C_BLOB({Méthode;}Var1 {;Var2;...;VarN})

Ces commandes servent à déclarer de façon explicite les variables simples que vous utilisez dans vos bases.

Leur utilisation se fait de la façon suivante :

C_BOOLEEN(Var)

Par cette directive, vous forcez le compilateur à créer une variable Var dont le type sera Booléen.

Dans le cas où une application comporte des directives de compilation, 4D Compiler les détecte et n'a pas à se livrer à une quelconque estimation.

Une directive a la priorité sur une déduction à partir d'une affectation ou d'une utilisation.

Note Les variables simples déclarées par la directive de compilation C_ENTIER sont considérés comme des Entiers longs, compris entre -2147483648 et +2147483647 comme pour les variables déclarées par la directive C_ENTIER LONG.

Quand utiliser des directives de compilation ?

Il ressort de ce que l'on a dit précédemment que les directives de compilation sont utiles dans deux cas :

- lorsque le compilateur ne peut pas déduire seul le type d'une variable,
- lorsque vous voulez éviter que le compilateur fasse des déductions.

Par ailleurs, utiliser des directives de compilation peut vous permettre de réduire le temps de compilation.

Nous allons maintenant envisager ces différents cas en détail.

Cas d'ambiguïté

Il arrive que le compilateur ne puisse pas déduire le type d'une variable, et cela pour plusieurs raisons. Il est impossible de recenser tous les cas de figure. Une chose est certaine, c'est qu'en cas d'impossibilité à compiler, 4D Compiler vous donnera la raison précise du phénomène ainsi que les moyens d'y remédier.

On peut cependant isoler trois causes majeures d'hésitation pour 4D Compiler : l'ambiguïté proprement dite, l'ambiguïté sur une déduction forcée, et l'impossibilité totale de déduire un type.

■ L'ambiguïté proprement dite

Dans ce cas, une directive de compilation n'est pas obligatoire si vous êtes rigoureux dans le choix des noms de vos variables. L'ambiguïté sur le nom de la variable est le cas que nous avons vu précédemment : le compilateur choisit la première variable qu'il rencontre et assigne arbitrairement à la suivante, de même nom mais de type différent, le type qu'il a précédemment attribué.

▼ Prenons un exemple simple :

dans une méthode A,
LaVariable := **Vrai**

dans une méthode B,
LaVariable := "La lune est verte"

Dans le cas où la méthode A est compilée avant la méthode B, le compilateur considérera que LaVariable := "La lune est verte" est un changement de type d'une variable précédemment rencontrée. Le compilateur vous signalera qu'il y a retypage. Il génère une erreur qu'il vous appartient de corriger.

Ne vous inquiétez pas, le compilateur ne transformera pas votre variable LaVariable := "La lune est verte" en variable booléenne sans vous demander ce que vous en pensez !

■ L'ambiguïté sur une déduction forcée

Il peut arriver que 4D Compiler déduise un type sur un objet qui ne convient pas à son utilisation finale. Dans ce cas, vous devrez typer explicitement vos variables à l'aide des directives de compilation.

▼ Exemple de cas d'ambiguïté lors de l'utilisation des listes de valeurs par défaut sur un objet :

Dans les formulaires, il est possible de spécifier une liste de valeurs par défaut pour les objets de type combo box, pop up menu, menu/liste déroulante, onglet, zone de défilement et liste déroulante à l'aide du

bouton d'édition "Valeurs" (voir à ce sujet le manuel *Mode Structure* de 4^e Dimension, chapitre "Travailler avec les champs et les objets actifs"). Les valeurs par défaut sont automatiquement chargées dans un tableau dont le nom est le même que celui de l'objet.

Dans le cas simple où l'objet n'est pas utilisé dans une méthode, 4D Compiler peut déduire son type sans ambiguïté et l'objet sera typé en tableau texte par défaut.

En revanche, dans le cas où vous devez initialiser la position de votre tableau pour son affichage dans le formulaire, vous pouvez écrire les instructions suivantes dans la méthode formulaire :

Au cas ou

: (**Evenement formulaire**=Sur chargement)

MonPopUp:=2

...

Fin de cas

C'est dans ce cas que l'ambiguïté apparaît : lors de l'analyse des méthodes, 4D Compiler déduira par défaut le type Réel pour la variable *MonPopUp*. Dans ce cas très précis, vous devez explicitement déclarer le tableau dans une méthode COMPILER (cf. paragraphe ["Typage assuré par vos soins", page 33](#)) ou dans la méthode formulaire :

Au cas ou

: (**Evenement formulaire**=Sur chargement)

TABLEAU TEXTE(MonPopUp;2)

MonPopUp:=2

...

Fin de cas

■ L'impossibilité totale de déduire un type

Dans ce cas, seule une directive de compilation peut orienter le compilateur. Le compilateur peut se trouver dans cette situation lorsqu'une variable est utilisée sans être déclarée et dans un cadre qui ne donne aucune information sur son type possible.

Le phénomène se produit principalement dans quatre cas :

- lorsque vous utilisez des pointeurs,
- lorsque vous utilisez une commande à syntaxe multiple,
- lorsque vous utilisez une commande 4D avec des paramètres optionnels de types différents,
- lorsque vous utilisez une méthode appelée via une URL.

Cas des pointeurs

Un pointeur étant un outil universel qui a donc pour première caractéristique la flexibilité, il est inutile d'espérer qu'il renvoie un type d'une manière ou d'une autre, hormis le sien propre.

Supposons que vous écriviez dans une méthode la séquence suivante :

```
LaVar1:=5,2                (1)
LePointeur:=->LaVar1      (2)
LaVar2:=LePointeur->       (3)
```

Bien que la ligne (2) définisse le type de la variable pointée par le pointeur LePointeur, LaVar2 n'est pas pour autant typée. Lors de la compilation, 4D Compiler peut reconnaître un pointeur, mais n'a aucun moyen de savoir sur quel type de variable il pointe. Il ne peut donc pas déduire le type de LaVar2 ; une directive de compilation du type C_REEL(LaVar2) est donc indispensable.

Cas des commandes à syntaxe multiple

Lorsque vous utilisez une variable associée à la fonction Année de, la variable ne peut être, compte tenu de la nature même de la fonction, que de type Date.

En revanche, prenons un cas extrême : la commande PROPRIETES CHAMP admet deux syntaxes :

```
PROPRIETES CHAMP(NoDeTable;NoDeChamp;Type;Longueur;Indexée)
PROPRIETES CHAMP(Pointeur_Champ;Type;Longueur;Indexée)
```

Lorsque vous utilisez cette commande, le compilateur ne peut pas deviner quelle syntaxe et quels paramètres vous avez choisis. Il vous appartient alors d'orienter le compilateur par une directive de compilation.

Cas des commandes 4D ayant des paramètres optionnels de types différents

Lorsque vous utilisez une commande 4D qui accepte plusieurs paramètres optionnels de différents types, 4D Compiler ne peut pas deviner quels paramètres ont été passés.

Par exemple, la commande INFORMATION ELEMENT admet deux paramètres optionnels. Le premier est de type Entier long, le second est de type Booléen.

La commande peut donc être utilisée comme ceci :

INFORMATION ELEMENT(liste;position;num;texte;sous-liste;déployé)

ou comme cela :

INFORMATION ELEMENT(liste;position;num;texte;déployé)

Vous devez donc utiliser des directives de compilation pour typer les paramètres optionnels passés à la commande (s'ils n'ont pas déjà été typés suite à leur utilisation dans un autre endroit de la base).

Cas des méthodes appelées via une URL

Si vous écrivez des méthodes appelées via une URL, il est nécessaire de déclarer explicitement la variable Texte \$1 dans vos méthodes, par l'intermédiaire de l'instruction C_TEXTE(\$1), dans le cas où vous n'utilisez pas \$1 dans la méthode. En effet, le compilateur ne peut pas deviner qu'une méthode 4D va être appelée via une URL.

Réduction du temps de compilation

Si toutes les variables utilisées dans votre base sont explicitement déclarées, il n'est pas nécessaire que 4D Compiler refasse tout le typage. Dans ce cas, vous pouvez lui demander d'effectuer uniquement la phase de traduction de vos méthodes. Ainsi, vous économiserez environ 50 % du temps de compilation.

Cas d'optimisation

Les directives de compilation peuvent vous aider à accélérer vos méthodes. Pour plus de précision à ce sujet, reportez-vous au [chapitre "Guide du typage", page 39](#).

Pour nous en tenir à un exemple simple à ce stade du manuel, imaginez que vous incrémentiez un compteur. Si vous n'avez pas déclaré la variable, 4D Compiler considérera par défaut qu'elle est de type Numérique (ou réel). Si vous prenez soin de préciser qu'il s'agit d'un Entier, l'exécution de la base compilée sera plus satisfaisante. En effet, un Réel occupe, sur PC par exemple, 8 octets en mémoire alors que si vous choisissez un type limitatif, Entier ou Entier long, le compteur n'en occupera que 4. Il est bien évident que l'incrémentation d'un compteur de 8 octets est plus longue que celle d'un compteur de 4 octets.

Où placer vos directives de compilation ?

Vous avez deux possibilités selon que vous voulez que 4D Compiler vérifie ou non votre typage.

Typage des variables par 4D Compiler

Si vous voulez que 4D Compiler vérifie votre typage, ou bien s'en charge lui-même, placer une directive de compilation est simple. Vous avez le choix entre deux possibilités, qui correspondent d'ailleurs à deux méthodes de travail :

- ou bien vous écrivez cette directive lors de la première utilisation de la variable, qu'il s'agisse d'une variable locale, process ou interprocess. La seule recommandation en la matière est que vous l'inscriviez bien à la première utilisation de la variable dans la première méthode exécutée.

Note Lors de la compilation, 4D Compiler prend les méthodes dans l'ordre de leur création, et non dans l'ordre dans lequel elles apparaissent dans l'Explorateur.

- ou bien, si vous êtes systématique, regroupez toutes vos variables process ou interprocess avec les directives afférentes dans la méthode base Sur ouverture ou une méthode appelée par la méthode base Sur ouverture.
Pour les variables locales, regroupez ces directives en tête de la méthode où elles sont utilisées.

Typage assuré par vos soins

Si vous voulez que 4D Compiler n'ait pas à vérifier votre typage, vous devez alors donner au logiciel les clés de l'identification de ses objets.

La convention à respecter est la suivante : les directives de compilation des variables process ou interprocess, ainsi que les paramètres, devront être placées dans une ou plusieurs méthodes dont le nom commence par COMPILER.

Vous pourrez ainsi nommer ces méthodes COMPILER, COMPILER1, COMPILERTyp...

Vous pouvez placer toutes les directives de compilation de la base dans une seule méthode. Pour des raisons de clarté et de facilité de maintenance de votre base, nous vous recommandons de classer ces directives selon les quatre catégories de variables pour lesquelles le compilateur a besoin d'informations initiales.

Ces quatre catégories sont :

- les variables interprocess,
- les variables process,
- les variables locales,
- les paramètres d'appel des méthodes.

Le compilateur a en effet besoin des types des paramètres reçus par une méthode afin de pouvoir compiler les appels à cette méthode. La déclaration de ces paramètres obéit à la syntaxe suivante :

Directive (nom de méthode; param)

▼ Exemple

C_BOOLEEN(*LaMéthode*;\$1)

La liste des directives de compilation, classée, vous est d'ailleurs donnée par le compilateur lui-même si vous le souhaitez. Pour plus de détails au sujet du fichier de typage, reportez-vous au paragraphe [“Le fichier de typage”](#), page 119.

Note Cette syntaxe n'est pas exécutable en mode interprété.

Afin d'assurer la compatibilité parfaite entre les bases compilées et les bases non compilées, vous pouvez choisir d'exécuter l'une ou l'autre de ces méthodes dans 4^e Dimension. Pour cela, reportez-vous au manuel *Langage* de 4^e Dimension.

Paramètres particuliers

■ **Les paramètres reçus par les méthodes base**

Ces paramètres sont typés par défaut par le compilateur, si la déclaration n'a pas été faite explicitement. Néanmoins, si vous les déclarez, la déclaration doit se faire à l'intérieur des méthodes base.

La déclaration de ces paramètres ne peut pas se faire dans une méthode COMPILER.

Exemple : “Sur connexion Web” reçoit deux paramètres \$1 et \$2 de type texte. En début de méthode, écrivez: C_TEXTE(\$1;\$2)

■ Les triggers

Le paramètre \$0 (Entier long), résultat d'un trigger, est typé par défaut par le compilateur, si la déclaration n'a pas été faite explicitement. Néanmoins si vous le déclarez, la déclaration doit se faire à l'intérieur du trigger.

La déclaration de ce paramètre ne peut pas se faire dans une méthode COMPILER.

■ Les objets acceptant l'événement formulaire "Sur glisser"

Le paramètre \$0 (Entier long), résultat d'un événement formulaire "Sur glisser", est typé par défaut par le compilateur, si la déclaration n'a pas été faite explicitement. Néanmoins si vous le déclarez, la déclaration doit se faire à l'intérieur de la méthode objet.

La déclaration de ce paramètre ne peut pas se faire dans une méthode COMPILER.

Note 4D Compiler n'initialise pas le paramètre \$0. Dès que vous utilisez l'événement formulaire Sur glisser, vous devez donc initialiser \$0. Par exemple :

```
C_ENTIER LONG($0)
Si (Evenement formulaire=Sur glisser)
    $0:=0
    ...
    Si ($TypeDeDonnées=Est une image)
        $0:=-1
    Fin de si
    ...
Fin de si
```

Une syntaxe particulière : C_ALPHA

La syntaxe de toutes les directives de compilation est extrêmement simple ; seule la commande C_ALPHA requiert une attention particulière puisqu'elle admet un paramètre supplémentaire : la longueur maximale de la chaîne.

C_ALPHA(Longueur;Var1 {;Var2;...;VarN})

Par définition, C_ALPHA porte sur des chaînes fixes, il est donc naturel de donner la longueur de cette chaîne. A cet égard, il convient de rappeler une différence de comportement entre une base interprétée et une base compilée.

Dans une base interprétée, la séquence suivante :

```
LaLongueur:=15  
C_ALPHA(LaLongueur; LaChaine)
```

serait parfaitement logique.

4^e Dimension interpréterait LaLongueur puis remplacerait LaLongueur par sa valeur dans la directive.

En revanche, le compilateur utilise cette commande durant le typage des variables et en dehors de toute affectation particulière. Il ne peut donc pas savoir que LaLongueur est égale à 15. Ne connaissant pas la longueur de la chaîne, il ne peut pas lui réserver de place dans la table des symboles. Par conséquent, dans l'optique d'une compilation, il convient d'utiliser une constante pour spécifier la longueur de la chaîne de caractères déclarée. Cette déclaration se fera donc de la façon suivante :

```
C_ALPHA(15;LaChaine)
```

Il en va de même pour la déclaration de tableaux d'alphas par la commande :

```
TABLEAU ALPHA(LaLongueur;LeTableau;Nombre d'éléments)
```

Le paramètre indiquant la longueur des chaînes du tableau sera une constante.

En revanche, vous pouvez utiliser les constantes 4D ou des valeurs hexadécimales pour spécifier la longueur des chaînes dans ces deux directives de compilation. Exemple :

```
C_ALPHA(Constante4D;LaChaine)  
TABLEAU ALPHA(Constante4D;LeTableau;2)  
C_ALPHA(0x000A;LaChaine)  
TABLEAU ALPHA(0x000A;LeTableau;2)
```

Note Ne confondez pas la longueur d'un champ Alphanumérique qui peut être au maximum de 80 caractères avec une variable Alphanumérique. La longueur d'une chaîne déclarée par la directive **C_ALPHA** ou appartenant à un **TABLEAU ALPHA** peut être comprise entre 1 et 255. La syntaxe de cette commande vous permet de déclarer plusieurs variables de même longueur en une seule ligne. Si vous souhaitez déclarer plusieurs chaînes fixes de longueur différente, il faudra le faire en plusieurs lignes.

Une liberté permise par 4D Compiler

Les directives de compilation lèvent toute ambiguïté sur les types et, pour le cas des chaînes alphanumériques, sur les longueurs. L'exigence de rigueur ne se fait pas pour autant intolérance.

S'il vous arrive d'utiliser un Numérique là où vous avez déclaré un Entier ou de manipuler une chaîne de 30 caractères là où vous en avez déclaré une de 10, 4D Compiler ne considère pas qu'il y a conflit et suit simplement vos directives. Ainsi, si vous écrivez :

```
C_ENTIER(vEntier)
vEntier:=2,5
```

4D Compiler ne verra pas un conflit de types de nature à empêcher la compilation et prendra automatiquement en compte la partie entière du nombre affecté (2 au lieu de 2,5).

De la même façon, s'il vous arrive de déclarer une chaîne plus courte que la chaîne que vous manipulez, 4D Compiler ne prend que le nombre de caractères déclarés. Ainsi, dans la séquence suivante :

```
C_ALPHA(10;LaChaine)
LaChaine:="Il fait très beau aujourd'hui"
```

4D Compiler prend en compte les 10 premiers caractères de la constante, soit "Il fait tr".

Conclusion

Ce chapitre vous a fourni l'analyse générale du comportement de 4D Compiler dans le repérage des types de variables. Pour une compréhension plus détaillée et plus directement orientée sur la pratique, nous vous conseillons de lire attentivement les deux chapitres suivants. Ils illustrent le propos de ce chapitre en présentant d'une manière systématique :

- un guide des conflits possibles de types et les manières de les éviter. C'est le [chapitre "Guide du typage", page 39](#).
- une liste des commandes de 4^e Dimension pour lesquelles il est important d'avoir toujours en tête la syntaxe appropriée dans l'optique de la compilation d'une application. C'est le [chapitre "Précisions de syntaxe", page 61](#).

3

Guide du typage

Ce chapitre décrit les principales causes de conflits de types sur les variables, ainsi que les manières de les éviter.

Conflits sur les variables simples

Les conflits de types simples peuvent se résumer comme suit :

- conflit entre deux utilisations,
- conflit entre une utilisation et une directive de compilation,
- conflit par retypage implicite,
- conflit entre deux directives de compilation.

Conflit entre deux utilisations

Comme nous l'avons vu précédemment, le conflit de types le plus simple est celui pour lequel un même nom de variable désigne deux objets différents.

Imaginez que dans une application, vous écriviez :
`LaVariable:=5`

et que, quelque part ailleurs, dans la même application, vous écriviez :
`LaVariable:=Vrai`

Vous générez un conflit de types. Le remède est simple : renommez l'une des deux variables.

Conflit entre une utilisation et une directive de compilation

Imaginez que dans une application, vous écriviez :
`LaVariable:=5`

et que, quelque part ailleurs, dans la même application, vous écriviez :
`C_BOOLEEN(LaVariable)`

Le compilateur, peignant d'abord les directives de compilation, fera de LaVariable un Booléen mais lorsqu'il découvrira :

```
LaVariable:=5
```

il signalera un conflit de types. Ici encore, le remède est simple : renommez votre variable ou modifiez la directive de compilation.

L'utilisation de variables de types différents dans une expression génère des incohérences. Le compilateur signale très logiquement les incompatibilités. Prenons un exemple simple. Vous écrivez :

```
vBooléen:=Vrai
`4D Compiler déduit que vBooléen est de type Booléen
C_ENTIER(<>vEntier)
`Déclaration d'un Entier par une directive de compilation
<>vEntier:=3
`Commande compatible avec la directive de compilation
LaVar:= <>vEntier+vBooléen
`Opération contenant des variables dont les types sont incompatibles
```

Conflit par retypage implicite

Certaines fonctions renvoient des variables d'un type bien précis. L'affectation du résultat d'une de ces variables à une variable déjà typée différemment provoquera un conflit de types si vous ne faites pas attention.

Dans une application interprétée, vous pouvez écrire :

```
No_Ident:=Demander("Numéro d'identification")
`No_Ident est de type Texte
Si(Ok=1)
  No_Ident:=Num(No_Ident)
  `No_Ident est de type Numérique
  CHERCHER([Personnes]Id= No_Ident)
Fin de si
```

Vous générez ici un conflit de type sur la troisième ligne. Le remède consiste à contrôler le comportement de la variable. Dans certains cas, vous aurez à créer des variables intermédiaires d'un nom différent. Dans d'autres cas, comme celui-ci en particulier, vous pouvez structurer différemment votre méthode :

```
No_Ident:=Num(Demander("Numéro d'identification"))
`No_Ident est de type Numérique
Si(Ok=1)
  CHERCHER([Personnes]Id=No_Ident)
Fin de si
```


Conflit entre deux directives de compilation

Déclarer deux fois la même variable par deux directives de compilation différentes constitue, bien sûr, un retypage. Si, dans la même base, vous écrivez :

```
C_BOOLEEN(LaVariable)
C_TEXTE(LaVariable)
```

Le compilateur est confronté à un dilemme et vous demande quelles étaient vos intentions. Le remède est simple : renommez l'une des deux variables.

Comme nous l'avons vu dans le [chapitre "Notions de base", page 23](#), il ne faut pas oublier que le conflit de types pour une directive **C_ALPHA** peut surgir si vous modifiez la longueur maximale de la chaîne de caractères.

Ainsi, si vous écrivez :

```
C_ALPHA(5;MaChaine)
MaChaine:="Fleur"
C_ALPHA(7;MaChaine)
MaChaine:="Bonjour"
```

le compilateur est dans une situation de conflit puisque dans la déclaration des variables de type Alphanumérique, il doit réserver un emplacement de taille adéquate.

Dans ce cas, le remède consiste à donner une directive de compilation qui prenne la longueur maximale, puisque par défaut, le compilateur acceptera une longueur inférieure.

Vous pouvez donc écrire :

```
C_ALPHA(7;LaChaine)
LaChaine:="Fleur"
LaChaine:="Bonjour"
```

Note Si vous aviez écrit deux fois **C_ALPHA**(7;LaChaine), c'est-à-dire :

```
C_ALPHA(7;LaChaine)
LaChaine:="Fleur"
C_ALPHA(7;LaChaine)
LaChaine:="Bonjour"
```

le compilateur l'accepterait tout à fait. C'est simplement redondant.

Note sur les variables locales

Les conflits de types pour les variables locales sont absolument identiques aux conflits de types pour les variables process et interprocess, à ceci près que ces conflits se déroulent dans un espace plus restreint.

Les conflits se jouent au niveau général de la base pour les variables process et interprocess.

Les conflits se jouent au niveau particulier de la méthode pour les variables locales. Vous ne pouvez donc pas écrire dans la même méthode :

```
$Temp:="Bonjour"
```

et puis plus loin

```
$Temp:=5
```

En revanche, vous pouvez écrire dans une méthode M1 :

```
$Temp:="Bonjour"
```

et dans une méthode M2 :

```
$Temp:=5
```

Conflits sur les variables tableau

Les conflits possibles pour un tableau ne portent jamais sur la taille du tableau. Comme dans 4^e Dimension non compilé, les tableaux sont gérés dynamiquement. La taille d'un tableau peut varier au fil des méthodes et vous n'avez pas, bien sûr, à déclarer une taille maximale pour un tableau.

Vous pouvez, en conséquence, dimensionner un tableau à zéro, ajouter ou retirer des éléments, en effacer le contenu. Dans l'optique de la compilation, et ce, dans une même méthode, pour un tableau local ou dans toute la base pour un tableau process ou interprocess, vous devez veiller aux points suivants :

- ne pas changer le type des éléments du tableau,
- ne pas changer le nombre de dimensions d'un tableau,
- dans le cas des tableaux Alpha, ne pas changer la longueur des chaînes de caractères.

Changer le type des éléments d'un tableau

Si vous déclarez un tableau comme étant un tableau d'Entiers, il doit rester un tableau d'Entiers pour toute la base. Il ne pourra jamais contenir, par exemple, des éléments de type Booléen.

Si, dans une application, vous écrivez :

```
TABLEAU ENTIER(LeTableau;5)
```

```
TABLEAU BOOLEEN(LeTableau;5)
```

le compilateur ne peut identifier pour vous le type de *LeTableau*. Renommez simplement l'un des deux tableaux.

Changer le nombre de dimensions d'un tableau

En version non compilée, il peut vous arriver de changer le nombre de dimensions d'un tableau.

Lorsque le compilateur établit sa table des symboles, il gère différemment les tableaux à une dimension et les tableaux à deux dimensions.

En conséquence, un tableau déclaré une fois comme étant un tableau à une dimension ne peut être re-déclaré ou utilisé comme un tableau à deux dimensions et vice versa.

Dans une même base, vous ne pouvez donc pas avoir :

```
TABLEAU ENTIER(LeTableau1;10)
```

```
TABLEAU ENTIER(LeTableau1;10;10)
```

En revanche, vous pouvez, évidemment, avoir dans la même application

```
TABLEAU ENTIER(LeTableau1;10)
```

```
TABLEAU ENTIER(LeTableau2;10;10)
```

Par ailleurs, vous pouvez parfaitement écrire :

```
TABLEAU BOOLEEN(LeTableau;5)
```

```
TABLEAU BOOLEEN(LeTableau;10)
```

Comme vous pouvez le noter dans nos exemples, c'est le nombre de dimensions d'un tableau qu'on ne peut changer en cours d'application et non la valeur des dimensions du tableau.

Note Un tableau à deux dimensions est en fait un ensemble de plusieurs tableaux à une dimension. Pour plus de précisions, reportez-vous à la section concernant les tableaux dans le manuel *Langage* de 4^e Dimension.

Cas des tableaux de chaînes fixes

Les tableaux de chaînes fixes, aussi appelés tableaux Alpha, suivent la même règle que les variables Alphanumériques et pour les mêmes raisons.

Si vous écrivez :

```
TABLEAU ALPHA(5;LeTableau;10)
```

```
TABLEAU ALPHA(10;LeTableau;10)
```

le compilateur détecte un conflit de longueur.

Le remède est simple : comme dans les chaînes Alpha, vous déclarez la longueur maximale. 4D Compiler gère automatiquement les longueurs inférieures.

Retypages implicites

Lors de l'utilisation des commandes COPIER TABLEAU, ENUMERATION VERS TABLEAU, TABLEAU VERS ENUMERATION, SELECTION VERS TABLEAU, SOUS SELECTION VERS TABLEAU, TABLEAU VERS SELECTION, VALEURS DISTINCTES, vous pouvez, volontairement ou involontairement, être conduit à des changements de type d'éléments ou de nombre de dimensions, ou pour un tableau Alpha, à des changements de longueur de chaîne. Vous vous retrouverez donc dans un des trois cas cités précédemment.

Le compilateur vous délivrera un message d'erreur et la correction que vous aurez à faire sera en général évidente.

Nous donnons des exemples de retypage implicite de tableaux dans le chapitre 4 de ce manuel, lorsque nous passons en revue ces commandes ([chapitre "Précisions de syntaxe", page 61](#)).

Tableaux locaux

Si vous souhaitez compiler une base de données qui utilise des tableaux locaux (tableaux visibles uniquement par les méthodes qui les ont créés), il est nécessaire de les déclarer explicitement dans 4D avant de les utiliser.

La déclaration explicite d'un tableau signifie l'utilisation d'une commande de type TABLEAU REEL, TABLEAU ENTIER, etc.

Par exemple, si une méthode génère un tableau local d'entiers contenant 10 valeurs, vous devez écrire au préalable la ligne suivante :
TABLEAU ENTIER(\$MonTableau;10)

Note Pour plus d'informations, reportez-vous au manuel *Langage de 4^e Dimension*.

Typage des variables dessinées dans les formulaires

Les variables dessinées dans un formulaire, qu'il s'agisse d'une case à cocher ou d'une zone externe, sont toutes des variables soit process, soit interprocess.

Dans une application en interprété, la question des types de ces variables ne se pose pas dans la pratique. Elle peut se poser, en revanche, dans l'optique d'une application compilée. Les règles du jeu sont cependant transparentes :

- soit vous avez typé votre variable,
- soit le compilateur lui attribue un type par défaut qui peut être défini dans le projet de compilation (voir le [chapitre "Fenêtres des options"](#) paragraphe "Type par défaut des boutons", page 98).

Variables considérées par défaut comme des Numériques

Les variables suivantes sont considérées comme des Numériques par défaut :

- Case à cocher
- Case a cocher 3D
- Bouton
- Bouton inversé
- Bouton invisible
- Bouton 3D
- Bouton image
- Grille de boutons
- Bouton radio
- Bouton radio 3D
- Radio image
- Menu image
- Menu déroulant hiérarchique

- Liste hiérarchique
- Règle
- Cadran
- Thermomètre

Note Les variables de type Règle, Cadran et Thermomètre seront toujours typées comme des Numériques, même si vous avez choisi l'option Entier long par défaut pour le type des boutons dans votre projet de compilation.

Pour ces variables, vous ne pouvez jamais vous trouver dans un conflit de types puisqu'elles ne peuvent avoir d'autres types que ce type-là, qu'on soit en interprété ou en compilé.

Les seuls conflits de types possibles sur l'une de ces variables viendraient du fait que le nom d'une variable serait le même que celui d'une autre variable placée à un autre endroit dans l'application. Dans ce cas, le remède consiste à renommer cette deuxième variable.

Variable Graphe

Une zone de graphe a automatiquement le type Graphe. Il ne peut jamais y avoir de conflit de type.

Les seuls conflits de type possibles sur une variable de type Graphe viendraient du fait que le nom de cette variable serait le même que celui d'une autre variable placée à un autre endroit dans l'application. Dans ce cas, le remède consiste à renommer cette deuxième variable.

Note Dans la version 5, le type Graphe était un type 4D. Dorénavant, le type Graphe est du type Entier long.

Variable Zone externe

Une zone externe est toujours un Entier long. Il ne peut jamais y avoir de conflit de types.

Les seuls conflits de types possibles sur une variable Zone externe viendraient du fait que le nom de cette variable serait le même que celui d'une autre variable placée à un autre endroit dans l'application. Dans ce cas, le remède consiste à renommer cette deuxième variable.

Variables considérées par défaut comme du Texte

Ces variables sont les suivantes :

- Variable non-saisissable,
- Variable saisissable,
- Liste déroulante,
- Menu/liste déroulante,
- Zone de défilement,
- Combo box,
- Pop-up Menu,
- Onglet.

Ces variables se divisent en deux catégories :

- les variables simples (variables saisissables et variables non-saisissables),
- les variables d’affichage de tableaux (listes déroulantes, menus/listes déroulantes, zone de défilement, pop-up menu, combo box, onglets).

Variables simples

Par défaut, ces variables reçoivent le type Texte.

Si elles sont utilisées dans une méthode objet ou une méthode formulaire, c’est le type que vous avez choisi qui leur sera attribué.

Vous n’avez aucun risque de conflit de types autre que celui qui serait généré par une attribution préalable du même nom à une autre variable.

Variables d’affichage de tableaux

De nombreuses variables vous servent à afficher des tableaux dans les formulaires. Si les valeurs par défaut ont été saisies au niveau des contrôles de saisie des variables en mode structure, il est conseillé de typer les variables correspondantes explicitement par une déclaration de type tableau (TABLEAU ALPHA, TABLEAU TEXTE...).

Questions de type autour des pointeurs

Si vous utilisez des pointeurs dans vos applications, vous avez pu profiter de la puissance et de la flexibilité de cet outil dans 4^e Dimension. Le compilateur en conserve intégralement les avantages.

Un pointeur peut pointer indifféremment sur une table, une variable ou un champ. Un même pointeur peut pointer sur des variables de type différent : veillez à ne pas générer des conflits artificiellement, en attribuant à une même variable des types différents.

Faites simplement attention à ne pas changer en cours de route le type de la variable sur laquelle pointe le pointeur en faisant, par exemple, une manipulation du type suivant :

```
LaVariable:=5,3  
LePointeur:=-> LaVariable  
LePointeur->:=6,4  
LePointeur->:=Faux
```

Dans ce cas de figure, votre pointeur dépointé est une variable Numérique. En affectant à cette variable une valeur booléenne, vous provoquez un conflit de types.

Si vous avez besoin dans une même méthode d'utiliser les pointeurs pour des propos différents, prenez soin de définir votre pointeur :

```
LaVariable:=5,3  
LePointeur:=-> LaVariable  
LePointeur->:=6,4  
LeBool:=Vrai  
LePointeur:=->LeBool  
LePointeur->:=Faux
```

Un pointeur n'a aucune existence propre. Il est toujours défini en fonction de l'objet qu'il représente.

C'est pourquoi le compilateur ne peut pas détecter des conflits de types générés par une utilisation sans contrôle des pointeurs. Vous n'aurez donc pas, en cas de conflit, de message d'erreur durant la phase de typage ou celle de compilation.

Cela ne veut pas dire que lorsque vous utilisez des pointeurs, vous travaillez sans filet. Le compilateur peut vérifier vos utilisations de pointeurs lorsque vous choisissez l'option de compilation appropriée, c'est à dire la compilation avec Contrôle d'exécution. Pour cela, reportez-vous au [chapitre "Les aides à la compilation"](#), page 109.

Plug-ins

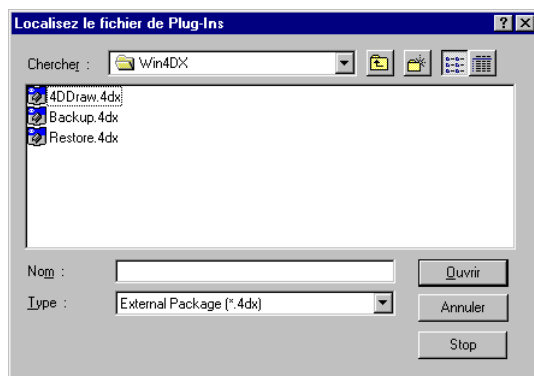
Généralités

Le compilateur a besoin, au moment de la compilation, des définitions des commandes des plug-ins utilisées dans la base à compiler, c'est-à-dire du nombre et du type des paramètres de ces commandes. Les risques d'erreur de typage sont inexistants à partir du moment où le compilateur trouve effectivement dans l'application ce que vous avez déclaré.

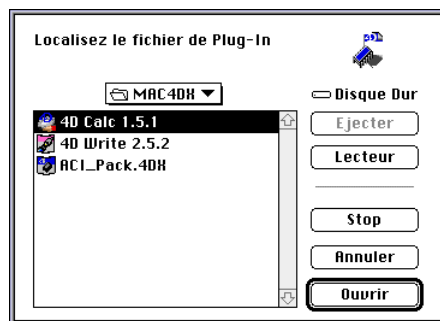
Placez vos plug-ins PC dans le répertoire WIN4DX et vos plug-ins Macintosh et Power Macintosh dans le dossier MAC4DX, à côté de votre structure. Le compilateur ne duplique pas le fichier mais tient compte des déclarations des commandes sans rien vous demander en supplément.

Si vos plug-ins sont placés ailleurs, 4D Compiler vous demande lors du typage de les localiser. Une boîte de dialogue d'ouverture de documents s'affiche.

Windows



MacOS



Cliquez sur le fichier de votre plug-in puis sur le bouton **Ouvrir**.

Le compilateur prend alors en compte la définition de votre commande. Là encore, il n'y a pas de risque de confusion au niveau du typage, si vos appels sont cohérents avec la déclaration de la commande. Il est préférable de passer à une commande de plug-in le nombre de paramètres qu'elle attend. En revanche, il est indispensable de recompiler votre base si vos commandes ont été modifiées.

Compilation multi-plate-forme

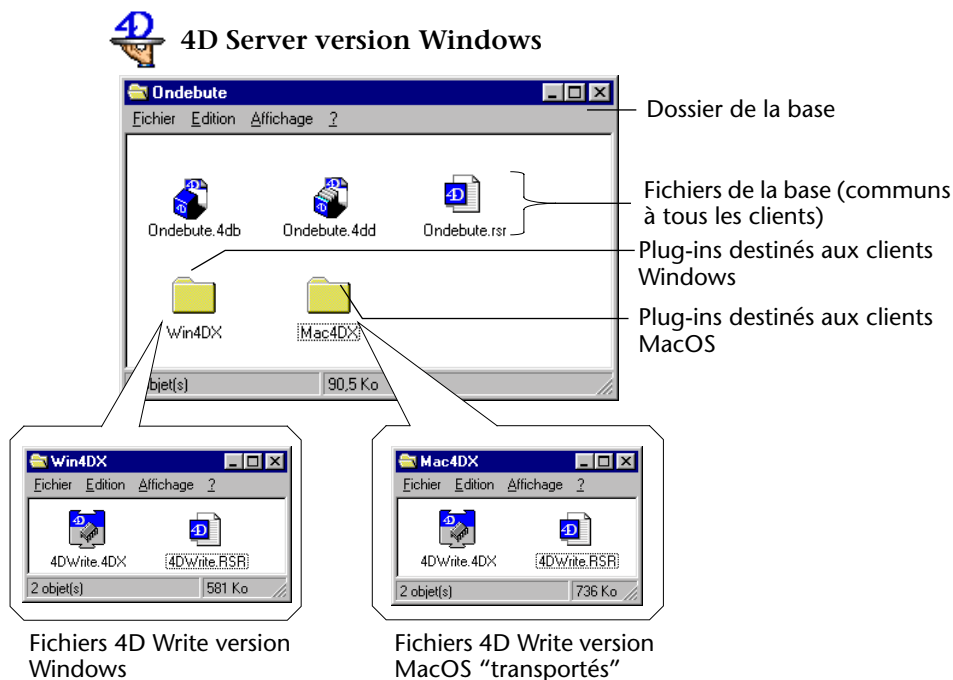
Compilation indépendante de plate-forme avec 4D Compiler version Windows

Si vous souhaitez compiler à partir de 4D Compiler version Windows une base contenant des plug-ins destinés à être exécutés sur une plate-forme MacOS avec 4D ou 4D Client version MacOS ou avec des 4D Client versions MacOS et Windows, vous devez procéder de la manière suivante :

- 1 Installez votre plug-in version MacOS sur votre Macintosh ou Power Macintosh.**
- 2 Transportez le fichier vers Windows, à l'aide de l'utilitaire "4D Transporter".**
Pour plus d'informations sur ce point, reportez-vous au manuel (électronique) de 4D Transporter.
- 3 Sous Windows, créez un nouveau dossier (ou "répertoire").**
- 4 Nommez ce dossier *Mac4DX*.**
- 5 Copiez les fichiers de votre plug-in "transportés" vers Windows dans ce dossier.**
Le plug-in "transporté" se compose des fichiers Nomplug-in.4DX et Nomplug-in.RSR.
- 6 Placez ce dossier au même niveau que le fichier de structure de la base que vous souhaitez compiler.**

Une configuration possible est résumée dans l'exemple suivant.

- Pour utiliser la base compilée avec 4D Server version Windows :



- Notes**
- Pour une compilation d'une base contenant des appels à des commandes 680xx contenues dans un fichier Proc.Ext ou Routines externes, il vous faut tout d'abord transporter votre fichier « Proc.Ext » avec 4D Transporter. Placez ensuite le fichier « Proc.esr » ainsi généré à côté de votre structure.
 - Pour plus d'informations sur l'indépendance de plate-forme de 4D Server, reportez-vous au *Manuel de référence* de 4D Server.

Compilation indépendante de plate-forme avec 4D Compiler version MacOS

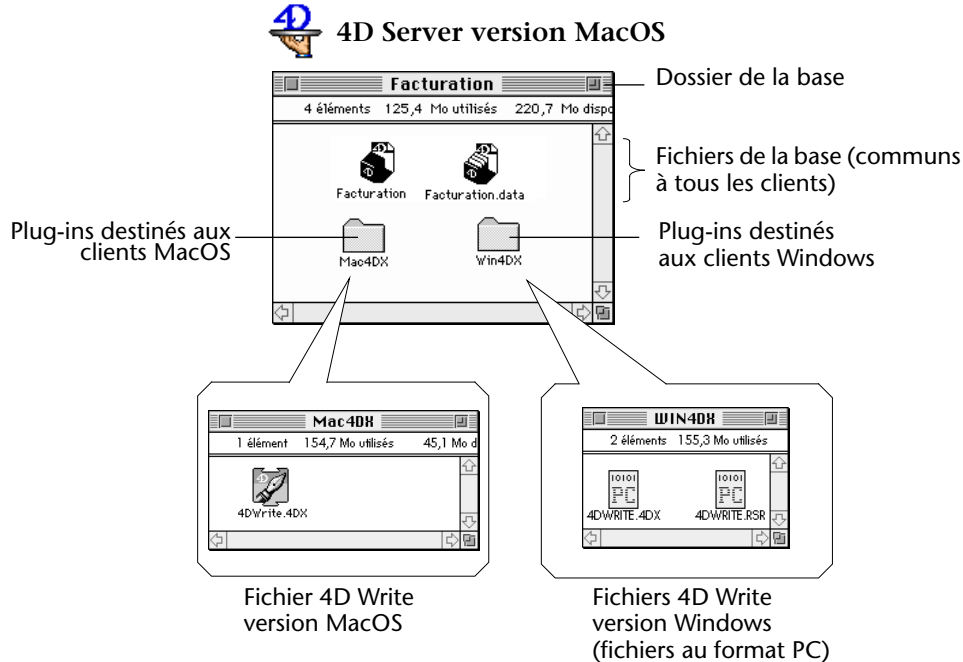
Si vous souhaitez compiler à partir de 4D Compiler version MacOS une base contenant des externes destinée à être exécutée sur une plate-forme Windows, avec 4D ou 4D Client version Windows ou 4D Client version MacOS et Windows, vous devez procéder de la manière suivante :

- 1 Recopiez les fichiers Windows de votre plug-in sur votre Macintosh ou Power Macintosh.**
4D Write version Windows se compose, par exemple, des fichiers "4DWrite.4DX" et "4DWrite.RSR".
- 2 Sous MacOS, créez un nouveau dossier et nommez-le Win4DX.**
- 3 Copiez les fichiers Windows de votre plug-in dans ce répertoire.**

4 Placez ce répertoire au même niveau que le fichier de structure de la base que vous souhaitez compiler.

Une configuration possible est résumée dans l'exemple suivant.

- Pour utiliser la base compilée avec 4D Server version MacOS :



- Note*
- Pour la compilation d'une base contenant des appels à des commandes 680XX contenues dans un fichier Proc.Ext ou Routines externes, il vous faut placer le fichier "Proc.Ext" à côté de votre structure.
 - Pour plus d'informations sur l'indépendance de plate-forme de 4D Server, reportez-vous au *Manuel de référence* de 4D Server.

Configuration nécessaire

Vous devez avoir placé les mêmes versions de vos plug-ins dans les dossiers MAC4DX et WIN4DX pour une compilation multi plate-forme.

- Si les mêmes commandes sont utilisées sous MacOS et sous Windows pour une même base, chaque numéro d'identification (ID) et le nombre de paramètres de chaque commande est le même dans le plug-in MacOS et le plug-in Windows.

- Pour un même plug-in, durant la compilation et l'exécution, le nom des plug-ins doit être le même dans le dossier MAC4DX et dans le dossier WIN4DX.
- ▼ **Exemple** : compilation sous MacOS ou Windows avec le plug-in 4D Write puis exécution de la base avec 4D Server sous MacOS ou sous Windows.
 - *Configuration MacOS pendant la compilation et l'exécution* :
 - NomDeLaBase
 - NomDeLaBase.data
 - Mac4DX
 - 4DWrite.4DX
 - Win4DX
 - 4DWrite.4DX
 - 4DWrite.RSR
 - *Configuration Windows pendant la compilation et l'exécution* :
 - NomBase.4DB
 - NomBase.RSR
 - NomBase.4DD
 - Mac4DX
 - 4DWrite.4DX
 - 4DWrite.RSR
 - Win4DX
 - 4DWrite.4DX
 - 4DWrite.RSR

Compilation et création d'un exécutable sous MacOS ou Windows

Lors de la création d'un exécutable avec 4D Compiler version MacOS, assurez-vous d'avoir au même niveau que la structure de la base le dossier MAC4DX ainsi que les fichiers Proc.ext ou Routines externes.

Lors de la création d'un exécutable avec 4D Compiler version Windows, assurez-vous que le répertoire WIN4DX est au même niveau que la structure de la base.

Dans le cas contraire, 4D Compiler affichera des messages d'erreur lors de la compilation.

Compilation avec les librairies OLE (Windows)

Si vous utilisez les librairies OLE dans votre base de données, effectuez avant la compilation les opérations suivantes :

- 1 **Créez un dossier nommé WIN4DX (s'il n'existe pas déjà).**
- 2 **Renommez le fichier OLETOOLS.DLL en fichier OLETOOLS.4DX.**

3 Placez les fichiers OLETOOLS.4DX et OLETOOLS.RSR dans le dossier WIN4DX.

Votre compilation peut alors s'effectuer.

Au lancement de votre base compilée, vous pouvez :

- soit garder le répertoire WIN4DX avec les deux fichiers OLETOOLS.4DX et OLETOOLS.RSR,
- soit utiliser les deux fichiers OLETOOLS.DLL et OLETOOLS.RSR dans le même répertoire que le fichier 4D.EXE.

Au lancement de votre application exécutable, vous devez conserver le répertoire WIN4DX avec les deux fichiers OLETOOLS.4DX et OLETOOLS.RSR.

Routines recevant des paramètres implicites

Certains plug-ins (4D Calc, 4D Write, 4D SQL Server...) utilisent des commandes qui provoquent l'appel implicite à des commandes 4^e Dimension.

- ▼ Prenons un exemple avec 4D Write. Vous disposez d'une commande appelée wr_APPELER SUR EVENEMENT. La syntaxe de cette commande est :

wr_APPELER SUR EVENEMENT(*LaZone;Événement;MéthodeÉvénement*)

Le dernier paramètre que vous passez à cette routine est le nom d'une méthode, que vous aurez vous-même écrite dans 4^e Dimension.

Cette méthode sera appelée par 4D Write chaque fois que l'événement attendu sera reçu et cette méthode recevra automatiquement les paramètres suivants :

Paramètres	Type	Description
\$0	Entier long	Prise en compte de l'événement
\$1	Entier long	Zone 4D Write
\$2	Entier long	Touche Maj.
\$3	Entier long	Touche Alt (Windows), Option (MacOS)
\$4	Entier long	Touche Ctrl (Windows), Commande (MacOS)
\$5	Entier long	Type d'événement qui a provoqué l'appel
\$6	Entier long	Valeur variant en fonction du paramètre Événement

Afin que le compilateur connaisse l'existence de ces paramètres et les prenne en compte, vous devez vous assurer qu'ils peuvent effectivement être typés, soit par une directive de compilation, soit parce que leur utilisation, suffisamment explicite, permet de déduire leur type.

Variables créées par des commandes de plug-ins

Les variables créées par le biais de commandes de plug-ins, et n'apparaissant pas dans le code de la structure, ne sont pas “vues” par 4D Compiler lors de la compilation.

MacOS

Pour que 4D Compiler tienne compte de ces variables, il faut créer une nouvelle ressource dans le fichier structure de votre base de données, à l'aide d'un éditeur de type ResEdit®. Cette ressource devra s'appeler VAR# si vos variables sont de type process et VAR◊ dans le cas de variables interprocess (le caractère “diamant” est obtenu par la combinaison de touches **Option-v**). Il suffit ensuite de déclarer vos variables à l'intérieur de ces ressources.

Windows

Pour que 4D Compiler tienne compte des variables VAR# et VAR◊ créées par des commandes de plug-ins, il faut avoir au préalable créé une ressource VAR# ou VAR◊. Ceci doit se faire sur une plate-forme MacOS à l'aide d'un éditeur de ressources de type ResEdit®.

Note Pour plus d'informations sur le traitement des commandes de plug-ins, reportez-vous à la documentation de *4D External Kit*.

Composants 4D

4^e Dimension et 4D Insider permettent (à compter de la version 6.7), de créer et de manipuler des *composants 4D*. Les composants s'apparentent à des bibliothèques d'objets 4D, dans lesquelles chaque objet dispose d'un attribut (“Privé”, “Protégé” ou “Public”) indiquant s'il sera visible, modifiable, etc. Pour plus d'informations sur la gestion des composants 4D, reportez-vous à la documentation de 4D Insider.

En principe, le développeur de composants 4D doit s'assurer que le composant est compilable et ne risque pas de générer de conflits. Cette possibilité ne peut toutefois être totalement exclue. En cas d'erreur de compilation provoquée par un objet appartenant à un composant,

4D Compiler affichera les informations suivantes, en fonction de l'attribut de l'objet :

- “Privé” : 4D Compiler ne fournit pas le nom de l'objet en cause, mais celui du composant 4D.
- “Protégé” ou “Public” : 4D Compiler désigne l'objet en cause, comme pour tout autre objet de la base (fonctionnement standard).

Manipulation des locales \$0...\$N et passation des paramètres

Les manipulations de ces locales suivent toutes les règles que nous avons déjà énoncées. De même que toutes les autres variables locales, elles ne peuvent être retypées en cours de méthode.

Dans ce paragraphe, nous abordons deux cas de figure où l'inattention pourrait vous conduire à des conflits de types :

- Lorsque vous avez en fait besoin d'un retypage. L'utilisation de pointeurs vous permet alors d'éviter les conflits de types.
- Lorsque vous avez besoin d'adresser des paramètres par indirection.

Utiliser les pointeurs pour éviter les retypages

Il n'est pas possible de retyper une variable. Il est, en revanche, tout à fait possible de faire pointer un pointeur successivement sur des variables de type différent.

Un exemple nous permet d'illustrer ce propos : écrivons une fonction qui nous renvoie l'occupation mémoire d'un tableau à une dimension suivant son type. Le résultat est un numérique dans tous les cas sauf deux : dans le cas des tableaux Texte et des tableaux Image, la taille mémoire occupée par le tableau dépend de valeurs inexprimables sous forme numérique (reportez-vous, dans le manuel *Langage* de 4^e Dimension, au chapitre concernant les Tableaux).

Dans le cas des tableaux Texte et des tableaux Image, nous renverrons comme résultat une chaîne de caractères. Cette fonction requiert un paramètre : un pointeur sur le tableau dont on veut connaître l'occupation mémoire.

Pour effectuer cette opération, vous avez le choix entre deux méthodes :

- ne travailler qu'avec des variables locales et ne pas vous soucier de leur type — mais dans ce cas, la méthode ne fonctionnera qu'en mode interprété.
- utiliser des pointeurs et alors travailler indifféremment en interprété et en compilé.

Fonction *OccupMém* en interprété seulement (exemple pour Macintosh)

```
$Taille:=Taille tableau($1->)
$Type:=Type($1->)
Au cas ou
:($Type=14)                ` Tableau Numérique
    $0:=8+($Taille*10)      ` $0 est un Numérique
:($Type=15)                ` Tableau Entier
    $0:=8+($Taille*2)
:($Type=16)                ` Tableau Entier Long
    $0:=8+($Taille*4)
:($Type=17)                ` Tableau Date
    $0:=8+($Taille*6)
:($Type=18)                ` Tableau Texte
    $0:=Chaine(8+($Taille*4))+("Somme des longueurs des textes")
                                ` $0 est un Texte
:($Type=19)                ` Tableau Image
    $0:=Chaine(8+($Taille*4))+("Somme des tailles des images")
                                ` $0 est un Texte
:($Type=20)                ` Tableau Pointeur
    $0:=8+($Taille*16)
:($Type=22)                ` Tableau Booléen
    $0:=8+($Taille/8)
```

Fin de cas

Dans la méthode ci-dessus, il y a changement de type de \$0 suivant les valeurs de \$1.

Fonction *OccupMém* en mode interprété et compilé (exemple pour Macintosh)

```
Ecrivons maintenant cette méthode en utilisant un pointeur :
$Taille:=Taille tableau($1->)
$Type:=Type($1->)
VarNum:=0
Au cas ou
:($Type=14)                ` Tableau Réel
    VarNum:=8+($Taille*10)  ` VarNum est un Numérique
:($Type=15)                ` Tableau Entier
    VarNum:=8+($Taille*2)
:($Type=16)                ` Tableau Entier Long
```

```
    VarNum:=8+($Taille*4)
:($Type=17)                                ` Tableau Date
    VarNum:=8+($Taille*6)
:($Type=18)                                ` Tableau Texte
    VarText:=Chaine(8+($Taille*4))+("+Somme des longueurs des textes")
:($Type=19)                                ` Tableau Image
    VarText:=Chaine(8+($Taille*4))+("+Somme des tailles des images")
:($Type=20)                                ` Tableau Pointeur
    VarNum:=8+($Taille*16)
:($Type=22)                                ` Tableau Booléen
    VarNum:=8+($Taille/8)
Fin de cas
Si (VarNum#0)
    $0:=->VarNum
Sinon
    $0:=->VarText
Fin de si
```

Il faut noter tout de même la différence entre ces deux séquences :

- dans le premier cas, le résultat de la fonction est la variable que l'on attendait,
- dans le second cas, le résultat de la fonction est un pointeur sur cette variable. Il vous appartient alors de simplement dépointer le résultat reçu.

Indirections sur les paramètres

4D Compiler gère la puissance et la souplesse de l'indirection sur les paramètres. En mode interprété, 4^e Dimension vous donne toute latitude concernant le nombre et le type des paramètres. Vous gardez en mode compilé cette même liberté à condition de ne pas introduire de conflit de types et de ne pas utiliser, dans la méthode appelée, plus de paramètres que vous en avez passés, ce qui est facile.

Afin de contourner un éventuel conflit, les paramètres adressés par indirection doivent tous être du même type.

Pour une bonne gestion de cette indirection, il est important de respecter la convention suivante : si tous les paramètres ne sont pas adressés par indirection, ce qui est le cas le plus fréquent, il faut que les paramètres adressés par indirection soient passés en fin de liste.

A l'intérieur de la méthode, l'adressage par indirection se fait sous la forme : `$(i)`, `i` étant une variable numérique. `$(i)` est appelé paramètre générique.

Illustrons notre propos par un exemple : écrivons une fonction qui prend des valeurs, fait leur somme et renvoie cette somme formatée suivant un format qui peut varier avec les valeurs.

A chaque appel à cette méthode, le nombre de valeurs à additionner peut varier. Il faudra donc passer comme paramètre à notre méthode les valeurs, en nombre variable, et le format, exprimé sous forme d'une chaîne de caractères.

Un appel à cette fonction se fera de la façon suivante :

Résultat:=*LaSomme*("##0,00";125,2;33,5;24)

La méthode appelante récupérera dans ce cas la chaîne : 182,70, somme des nombres passés, formatée suivant le format spécifié. D'après ce que l'on a vu plus haut, les paramètres de la fonction doivent être passés dans un ordre précis : le format d'abord et ensuite les valeurs, dont le nombre peut varier d'un appel à l'autre.

Examinons maintenant la fonction que nous appelons *LaSomme* :

\$Somme:=0

Boucle(\$i;2;Nombre de parametres)

\$Somme:=\$Somme+\${\$i}

Fin de boucle

\$0:=Chaine(\$Somme;\$1)

Cette fonction pourra être appelée de diverses manières :

Résultat:=*LaSomme*("##0,00";125,2;33,5;24)

Résultat:=*LaSomme*("000";1;18;4;23;17)

De même que pour les autres variables locales, la déclaration du paramètre générique par directive de compilation n'est pas obligatoire. Si elle est nécessaire (cas d'ambiguïté ou d'optimisation), elle se fait avec la syntaxe suivante :

C_ENTIER(\${4})

La commande ci-dessus signifie que tous les paramètres à partir du quatrième (inclus) seront adressés par indirection. Ils seront tous de type Entier. Les types de \$1, \$2 et \$3 pourront être quelconques. En revanche, si vous utilisez \$2 par indirection, le type utilisé sera le type générique. Il sera donc de type Entier, même si pour vous, par exemple, il était de type Réel.

Note Le compilateur utilisant cette commande durant le typage, le nombre, dans la déclaration, doit toujours être une constante et jamais une variable.

Variables réservées et constantes

Des variables et des constantes de 4^e Dimension ont un type et une identité fixés par le compilateur.

On ne peut donc créer une nouvelle variable, une méthode, une fonction ou une commande de plug-in portant le nom d'une de ces variables ou d'une de ces constantes. Bien entendu, vous pouvez tester leur valeur et les utiliser comme auparavant.

Variables système

Voici la liste complète des variables système de 4^e Dimension accompagnées de leur type.

Variable	Type
<i>OK</i>	Entier long
<i>Document</i>	Alpha (Chaîne fixe) : 255
<i>FldDelimit</i>	Entier long
<i>RecDelimit</i>	Entier long
<i>Error</i>	Entier long
<i>MouseDown</i>	Entier long
<i>KeyCode</i>	Entier long
<i>Modifiers</i>	Entier long
<i>MouseX</i>	Entier long
<i>MouseY</i>	Entier long
<i>MouseProc</i>	Entier long

Variables des états semi-automatiques

Lorsqu'on crée une colonne calculée dans un état, 4^e Dimension crée automatiquement une variable C1 pour la première, C2, C3... pour les autres. Généralement, cette création est faite de façon transparente pour vous.

Dans le cas où vous utiliseriez ces variables dans des méthodes, souvenez-vous que, comme les autres variables, les variables C1, C2... ne peuvent être retypées.

Constantes 4D

La liste des constantes peut être consultée dans le manuel *Langage* de 4D. Vous pouvez également les visualiser dans l'Explorateur, en mode Structure.

4

Précisions de syntaxe

4D Compiler suit la syntaxe habituelle des commandes 4^e Dimension et, en ce sens, ne vous demande aucun comportement particulier dans l'optique de la compilation.

Cela étant, nous consacrons ce chapitre à certains rappels et quelques précisions :

- Certaines commandes influant sur le type d'une variable peuvent, si vous n'y prêtez pas attention, vous conduire à des conflits de type.
- Certaines commandes admettant des syntaxes ou des paramètres différents, il est préférable de savoir ce qu'il est plus approprié de choisir.

Ces commandes sont regroupées thématiquement, par ordre alphabétique.

Chaînes de caractères

Code ascii (*LaChaine*)

Pour les routines opérant sur les chaînes, seule la fonction Code ascii réclame une attention plus particulière.

Lorsque vous travaillez en mode interprété, vous pouvez indifféremment passer une chaîne non vide ou vide à cette fonction.

En compilé, vous ne pouvez pas passer une chaîne vide.

Si vous le faites, le compilateur ne peut détecter une erreur à la compilation si l'argument passé à Code ascii est une variable.

Communications

ENVOYER VARIABLE(*LaVariable*)

RECEVOIR VARIABLE(*LaVariable*)

Ces deux commandes permettent d'écrire et de relire des variables envoyées sur disque. On passe des variables en paramètres à ces commandes.

Souvenez-vous simplement qu'il faudra toujours relire une variable d'un type dans une variable de même type.

Supposons que vous souhaitiez envoyer une liste de variables dans un fichier. Afin de ne pas prendre de risque de changement de type par inattention, nous vous recommandons d'adopter une méthode de travail simple qui consiste à indiquer en début de liste le type des variables envoyées. Ainsi, lorsque vous relirez ces variables, vous commencerez toujours par récupérer cet indicateur dont vous connaissez le type.

Ensuite, vous appellerez RECEVOIR VARIABLE en toute connaissance de cause par l'intermédiaire d'un Au cas ou.

▼ Exemple

```
REGLER SERIE(12;"LeFichier")
  Si (OK=1)
    $Type:=Type([Client]CA_Cumulé)
    ENVOYER VARIABLE($Type)
    Boucle($i;1;Enregistrements trouves)
      $CA_Envoi:=[Client]CA_Cumulé
      ENVOYER VARIABLE($CA_Envoi)
    Fin de boucle
  Fin de si
REGLER SERIE(11)
REGLER SERIE(13;"LeFichier")
  Si (OK=1)
    RECEVOIR VARIABLE($Type)
    Au cas ou
      :($Type=0)
        RECEVOIR VARIABLE($LaChaine)
        `Traitement de la variable reçue
      :($Type=1)
        RECEVOIR VARIABLE($LeRéel)
        `Traitement de la variable reçue
```

```

:($Type=2)
RECEVOIR VARIABLE($LeTexte)
  `Traitement de la variable reçue
Fin de cas
Fin de si
REGLER SERIE(11)

```

Note La liste des valeurs retournées par la fonction Type est fournie dans le manuel *Langage* de 4^e Dimension.

Définition structure

Champ (*Ptr_Champ*) ou (*NoDeTable;NoDeChamp*)

Table(*Ptr_Table*) ou (*Ptr_Champ*) ou (*NoDeTable*)

Dans l'optique du compilateur, ces deux commandes ne comportent rien de spécifique.

Nous appelons simplement votre attention sur un cas pratique : ces deux commandes retournent des résultats de type différent suivant le paramètre qui leur est passé :

- si vous passez un pointeur à la fonction Table, le résultat retourné sera de type Numérique.
- si vous passez un Numérique à la fonction Table, le résultat retourné sera de type Pointeur.

On comprend aisément que ces deux fonctions ne suffisent pas au compilateur pour déterminer le type du résultat. Dans ce cas, pour qu'il n'y ait aucune ambiguïté, utilisez une directive de compilation.

Documents système

Nous rappellerons simplement que la référence d'un document renvoyée par les fonctions Ouvrir document, Ajouter a document et Creer document est de type Heure.

Fonctions mathématiques

Modulo (*LaValeur;Diviseur*)

L'expression "25 modulo 3" peut s'écrire de deux façons différentes dans 4^e Dimension :

LaVariable:=**Modulo**(25;3)

ou

LaVariable:=25%3

Il existe pour le compilateur une différence entre ces deux écritures :

Modulo s'applique à tous les types de Numériques tandis que l'opérateur % s'applique exclusivement aux Entiers et Entiers longs (si les opérandes de l'opérateur % dépassent les limites des Entiers longs, le résultat renvoyé sera probablement faux).

Interruptions

APPELER 4D

APPELER SUR EVENEMENT (*LaMéthode*) ; {*NomProcess*})

APPELER SUR PORT SERIE (*LaMéthode*)

STOP

**APPELER SUR
EVENEMENT et
APPELER SUR PORT
SERIE**

Pour la gestion des interruptions, le langage de 4^e Dimension dispose de la commande APPELER 4D.

Cette commande devra être utilisée lorsque vous vous servirez des commandes APPELER SUR EVENEMENT et APPELER SUR PORT SERIE.

On pourrait définir cette commande comme une directive de gestion des événements.

Seul le noyau de 4^e Dimension peut détecter un événement Système (clic souris, action sur le clavier...). Dans la plupart des cas, des appels au noyau sont lancés par le code compilé lui-même, de façon transparente pour vous.

En revanche, dans le cas où vous attendez un événement sans rien faire, comme dans une boucle d'attente, il est bien évident qu'aucun appel n'est effectué.

▼ Exemple sous Windows

```
`Méthode projet ClicSouris
```

```
Si (MouseDown=1)
```

```
  <>vTest:=Vrai
```

```
    ALERTE("Quelqu'un a cliqué avec la souris")
```

```
Fin de si
```

```
Méthode projet Attente
```

```
<>vTest:=Faux
```

```
APPELER SUR EVENEMENT("ClicSouris")
```

```
Tant que(<>vTest=Faux)           `Boucle d'attente de l'événement
```

```
Fin tant que
```

```
APPELER SUR EVENEMENT("")
```

Dans ce cas, vous ajouterez la directive APPELER 4D de la façon suivante :

```
`Méthode projet Attente
```

```
<>vTest:=Faux
```

```
APPELER SUR EVENEMENT("ClicSouris")
```

```
Tant que(<>vTest=Faux)
```

```
  APPELER 4D
```

```
    `Appel au noyau pour discerner un événement
```

```
Fin tant que
```

```
APPELER SUR EVENEMENT("")
```

Note La commande APPELER SUR PORT SERIE n'a plus aucune utilité depuis la version 5 de 4^e Dimension et n'a été conservée que pour des raisons de compatibilité. En effet, vous pouvez reproduire le comportement de cette commande en lui apportant la dimension du multiprocess.

STOP

Cette commande ne doit être utilisée que dans des méthodes projet d'interception d'erreurs.

Cette commande fonctionne comme dans 4^e Dimension, sauf dans une méthode ayant été appelée par l'une des commandes suivantes : EXECUTER, APPLIQUER A SELECTION et APPLIQUER A SOUS SELECTION. Il convient d'éviter cette situation.

Tableaux

Sept routines de 4^e Dimension sont utilisées par le compilateur pour déterminer le type d'un tableau. Il s'agit de :

COPIER TABLEAU(*TableauSource*; *TableauDestination*)

SELECTION VERS TABLEAU(*LeChamp*; *LeTableau*)

TABLEAU VERS SELECTION(*LeTableau*; *LeChamp*)

SOUS SELECTION VERS TABLEAU(*Début*; *Fin*; *LeChamp*; *LeTableau*)

ENUMERATION VERS TABLEAU(*LaListe*; *LeTableau*; {*ItemRefs*})

TABLEAU VERS ENUMERATION(*LeTableau*; *LaListe*; {*ItemRefs*})

VALEURS DISTINCTES(*LeChamp*; *LeTableau*)

COPIER TABLEAU

COPIER TABLEAU admet deux paramètres de type Tableau.

Lorsque le compilateur rencontre cette commande pendant le typage et que l'un des paramètres tableau n'est pas déclaré ailleurs, le compilateur déduit le type du tableau non déclaré suivant le type de celui qui l'est.

Cette déduction est faite dans les deux cas suivants :

- le tableau typé ailleurs est le premier paramètre. Le compilateur donne au second tableau le type du premier.
- le tableau déclaré est le second paramètre. Dans ce cas, le compilateur donne au premier tableau le type du second.

Le compilateur étant rigoureux sur les types, un COPIER TABLEAU ne peut se faire que d'un tableau d'un type vers un tableau de même type.

En conséquence, si vous souhaitez faire une copie d'un tableau d'éléments de types voisins, c'est-à-dire les Entiers, Entiers longs et Numérique ou les Textes et les Alphas ou les Alphas de longueurs différentes, vous devrez le faire élément par élément.

- ▼ Imaginez que vous vouliez faire une copie d'un tableau d'Entiers vers un tableau de Numériques. Procédez comme suit :

```
$Taille:=Taille tableau(TabEntier)
```

```
TABLEAU REEL(TabRéel;$Taille)
```

```
`Donnons la même taille au tableau de Réels qu'au tableau d'Entiers
```

```
Boucle($i;1;$Taille)
```

```
TabRéel{$i}:=TabEntier{$i}
```

```
`Recopions chacun des éléments
```

```
Fin de boucle
```

Note Souvenez-vous que vous ne pouvez changer le nombre de dimensions d'un même tableau en cours de route. Vous provoqueriez une erreur de typage en copiant un tableau à une dimension dans un tableau à deux dimensions.

SELECTION VERS TABLEAU, TABLEAU VERS SELECTION, VALEURS DISTINCTES, SOUS SELECTION VERS TABLEAU

De même que pour 4^e Dimension en mode interprété, ces quatre commandes n'exigent pas de déclaration de tableau. Le tableau non déclaré recevra le même type que le champ spécifié dans la commande.

Si vous écrivez :

```
SELECTION VERS TABLEAU([LaTable]ChampEntier;LeTableau)
```

le type de LeTableau sera donc Tableau d'Entiers à une dimension.

Dans le cas où le tableau a déjà été déclaré, veillez à ce que les champs soient du même type. Bien qu'Entier, Entier long et Réel soient des types voisins, vous ne pouvez pas supposer qu'il soient équivalents.

Vous avez en revanche plus de latitude lorsqu'il s'agit des types Texte et Alpha. Par défaut, lorsqu'un tableau n'a pas été déclaré au préalable et que vous appliquez l'une de ces commandes avec pour paramètre un champ de type Alpha, le type attribué au tableau sera Texte.

Si le tableau a été déclaré auparavant comme étant de type Alpha ou de type Texte, ces commandes respecteront vos directives.

Il en est de même dans le cas des champs de type Texte : vos directives sont prioritaires.

Souvenez-vous que les commandes SELECTION VERS TABLEAU, SOUS SELECTION VERS TABLEAU, TABLEAU VERS SELECTION et VALEURS DISTINCTES ne s'appliquent qu'à des tableaux à une dimension.

La commande **SELECTION VERS TABLEAU** présente aussi une seconde syntaxe :

SELECTION VERS TABLEAU(LaTable;LeTableau)

Dans ce cas, la variable LeTableau sera de type Tableau d'Entiers longs. Il en est de même pour la commande **SOUS SELECTION VERS TABLEAU**.

ENUMERATION VERS TABLEAU, TABLEAU VERS ENUMERATION

Ces commandes ne concernent que deux types de tableaux :

- les tableaux Alpha à une dimension,
- les tableaux Texte à une dimension.

Ces deux commandes n'exigent pas la déclaration du tableau qui leur est passé en paramètre. Par défaut, un tableau non déclaré recevra le type Texte. Si le tableau a été déclaré auparavant comme étant de type Alpha ou de type Texte, ces commandes respecteront vos directives.

Utilisation des pointeurs dans les commandes s'appliquant aux tableaux

D'après ce que nous avons déjà vu au sujet des pointeurs, le compilateur ne peut pas, durant le typage ou la compilation, détecter un conflit de type dans le cas où vous utiliseriez des pointeurs dépointés comme paramètre de commande de déclaration d'un tableau. Si vous écrivez :

SELECTION VERS TABLEAU([LaTable]LeChamp;LePointeur->)

où *LePointeur->* représente un tableau, le compilateur ne peut vérifier que le type du champ et du tableau sont identiques. Il vous appartient d'éviter alors ce type de conflit.

Pour cela, le compilateur vous fournit une aide précieuse : les **Warnings**. Chaque fois qu'il rencontre une routine de déclaration de tableau dans laquelle l'un des paramètres est un pointeur, il vous délivre un message vous invitant à la vigilance.

Tableaux locaux

Si vous souhaitez compiler une base de données qui utilise des tableaux locaux (tableaux visibles uniquement par les méthodes qui les ont créés), il est nécessaire de les déclarer explicitement dans 4D avant de les utiliser.

La déclaration explicite d'un tableau signifie l'utilisation d'une commande de type **TABLEAU REEL**, **TABLEAU ENTIER**, etc.

Par exemple, si une méthode génère un tableau local d'entiers contenant 10 valeurs, vous devez écrire au préalable la ligne suivante :
TABLEAU ENTIER(\$MonTableau;10)

Note Pour plus d'informations, reportez-vous au manuel *Langage* de 4^e Dimension.

Langage

Pointeur vers(*NomVariable*)

Type (*Objet*)

EXECUTER(*LaMéthode*)

TRACE

PAS DE TRACE

Pointeur vers

Pointeur vers est une fonction qui retourne un pointeur sur le paramètre que vous lui avez passé. Supposons que vous vouliez initialiser un tableau de pointeurs. Chacun des éléments de ce tableau pointe vers une variable donnée. Ces variables sont au nombre de douze et nous les appelons V1, V2, ...V12.

Vous pourriez écrire :

TABLEAU POINTEUR(Tab;12)

Tab{1}:=>V1

Tab{2}:=>V2

...

Tab{12}:=>V12

Vous pouvez aussi écrire :

TABLEAU POINTEUR(Tab;12)

Boucle(\$i;1;12)

Tab{\$i}:=**Pointeur vers**("V"+**Chaine**(\$i))

Fin de boucle

A la fin de l'exécution de cette séquence, vous récupérez un tableau de pointeurs dont chaque élément pointe sur une variable *Vi*.

Ces deux séquences sont bien entendu compilables. Toutefois, si les variables V1 à V12 ne sont pas utilisées explicitement ailleurs dans la base, le compilateur ne pourra pas les typer. Il vous faut donc les nommer ailleurs explicitement.

Cette déclaration explicite peut se faire de deux façons :

- soit en déclarant *V1, V2, ...V12* par une directive de compilation :
C_ENTIER LONG(V1;V2;V3;V4;V5;V6;V7;V8;V9;V10;V11;V12)
- soit en affectant ces variables dans une méthode :
V1:=0
V2:=0
...
V12:=0

Type

Type(*Objet*)

Chaque variable de la base compilée n'ayant qu'un seul type, cette fonction peut sembler d'un intérêt limité. Elle est cependant très précieuse lorsqu'on travaille avec des pointeurs. En effet, il peut être nécessaire de connaître le type de la variable pointée par un pointeur, la souplesse des pointeurs faisant que l'on ne sait pas toujours sur quoi ils pointent.

EXECUTER

Cette commande, historique dans 4^e Dimension, présente des avantages en mode interprété qui n'ont pas grand sens en mode compilé.

En effet, en mode compilé, le nom de la méthode passé en paramètre à cette commande sera simplement interprété. Vous ne bénéficierez donc pas de tous les avantages apportés par le compilateur, sans compter que la syntaxe de votre paramètre ne pourra pas être vérifiée.

De surcroît, vous ne pouvez pas lui passer des variables locales comme paramètres.

On peut avantageusement remplacer un EXECUTER par une série d'instructions. Nous vous en donnons ici deux exemples.

Soit la séquence suivante :

i:= FctFormulaire

EXECUTER("FORMULAIRE ENTREE (Formulaire"+Chaine(i)+")")

Elle peut être remplacée par :

i:=FctFormulaire

VarFormulaire:="Formulaire"+Chaine(i)

FORMULAIRE ENTREE(VarFormulaire)

Examinons maintenant un deuxième cas :

```
$Num:=ChoixImprimante  
EXECUTER("Impri"+$Num)
```

Ici, l'EXECUTER peut être facilement remplacé par un Au cas ou :

```
Au cas ou  
  : ($Num=1)  
    Impri1  
  : ($Num=2)  
    Impri2  
  : ($Num=3)  
    Impri3
```

Fin de cas

La commande EXECUTER peut toujours être très avantageusement remplacée. En effet, la méthode à exécuter est prise dans la liste des méthodes projet de la base ou des commandes 4^e Dimension, qui sont en nombre limité. En conséquence, il est toujours possible de remplacer la commande EXECUTER soit par un Au cas ou, soit par une autre commande.

TRACE, PAS DE TRACE

Ces deux commandes, précieuses en mode interprété, n'ont pas de fonction en mode compilé. Vous pouvez cependant les laisser dans vos méthodes : elles seront simplement ignorées par le compilateur.

Variables

Indefinie(*LaVariable*)

ECRIRE VARIABLES(*NomduDoc*;*Variable1*{; *Variable2*...})

LIRE VARIABLES(*NomduDoc*;*Variable1*{; *Variable2*...})

Effacer variable(*LaVariable*)

Indefinie

Compte tenu du processus de typage par 4D Compiler, une variable ne peut à aucun moment être indéfinie en mode compilé. En effet, toutes les variables ont une existence dès la fin de la compilation. La fonction Indefinie retourne donc toujours Faux, quel que soit le paramètre que vous lui passez.

Note Pour savoir si votre application tourne en mode compilé, utilisez la fonction Application compilee.

ECRIRE VARIABLES, LIRE VARIABLES

En mode interprété, après l'exécution d'un LIRE VARIABLES, vous pouvez savoir si le document existe en testant si l'une des variables, théoriquement lue, est indéfinie ou non. Ceci n'est bien évidemment plus possible avec le compilateur, puisque la fonction Indefinie renvoie toujours Faux.

- La méthode la plus simple pour réaliser cette opération en mode interprété comme en mode compilé, est la suivante :
- 1 Initialisez les variables que vous allez recevoir à une valeur dont vous êtes sûr qu'elles ne sont pas initialisées.**
- 2 Après le LIRE VARIABLES, comparez l'une des variables reçues à la valeur d'initialisation.**

La méthode s'écrit alors de la façon suivante :

```
Var1:="xxxxxx"
  ` "xxxxxx" est une valeur qui ne peut pas être ramenée par un
  ` LIRE VARIABLES
Var2:="xxxxxx"
Var3:="xxxxxx"
Var4:="xxxxxx"
LIRE VARIABLES("LeDocument";Var1;Var2;Var3;Var4)
Si(Var1="xxxxxx")
  ` Le document n'a pas été trouvé
  ...
Sinon
  ` Le document a été trouvé
  ...
Fin de si
```

EFFACER VARIABLE

Cette routine admet deux syntaxes différentes en mode interprété :

```
EFFACER VARIABLE(LaVariable)
```

```
EFFACER VARIABLE("a")
```

En mode compilé, la première syntaxe, EFFACER VARIABLE(*LaVariable*), provoque non la destruction de la variable, mais sa réinitialisation (remise à zéro pour un numérique, chaîne vide pour une chaîne de caractères ou un texte...), aucune variable ne pouvant être indéfinie en mode compilé.

En conséquence, EFFACER VARIABLE ne libère pas de mémoire en mode compilé sauf dans quatre cas : les variables de type Texte, Image, BLOB et les tableaux.

Pour un tableau, EFFACER VARIABLE équivaut à une nouvelle déclaration du tableau ou les tailles sont remises à zéro.

Pour un tableau LeTableau dont les éléments sont de type Entier, EFFACER VARIABLE(LeTableau) équivaut à l'une des expressions suivantes :

TABLEAU ENTIER(LeTableau;0)

`s'il s'agit d'un tableau à une dimension

TABLEAU ENTIER(LeTableau;0;0)

`s'il s'agit d'un tableau à deux dimensions

La seconde syntaxe, EFFACER VARIABLE ("a"), n'est pas compatible avec le compilateur puisque celui-ci accède aux variables non par leur nom, mais par leur adresse mémoire.

Précisions concernant l'utilisation de pointeurs

Les commandes suivantes ont un point commun : elles admettent toutes un premier paramètre [LaTable] facultatif alors que le second paramètre peut être un pointeur.

ADJOINDRE ELEMENT	FORMULAIRE ENTREE
ALLER A ENREGISTREMENT	FORMULAIRE SORTIE
ALLER DANS SELECTION	GRAPHE SUR SELECTION
APPLIQUER A SELECTION	IMPRIMER ETIQUETTES
CHARGER ENSEMBLE	IMPRIMER LIGNE
CHERCHER	LECTURE ASCII
CHERCHER DANS SELECTION	LECTURE DIF
CHERCHER PAR FORMULE	LECTURE SYLK
CHERCHER PAR FORMULE DANS SELECTION	LIEN RETOUR
COPIER SELECTION	NOMMER ENSEMBLE
DEPLACER SELECTION	REDUIRE SELECTION
DIALOGUE	ENLEVER ELEMENT
ECRITURE ASCII	TRIER
ECRITURE DIF	TRIER PAR FORMULE
ECRITURE SYLK	UTILISER PARAMETRES IMPRESSION
ENSEMBLE VIDE	VERROUILLE PAR
ETAT	

Il est parfaitement possible en mode compilé de garder ce caractère optionnel au paramètre [LaTable].

Le compilateur fait toutefois une supposition dans le cas où le premier paramètre passé à l'une de ces commandes est un pointeur. Ne pouvant pas savoir sur quoi pointe ce pointeur, il suppose qu'il s'agit d'un pointeur de Table.

Prenons par exemple le cas de la commande CHERCHER dont la syntaxe est la suivante :

```
CHERCHER({LaTable}; LaMéthode;{*})
```

où le premier élément de la méthode doit être un champ.

Si vous écrivez :

```
CHERCHER(LePtrChamp->=Vrai)
```

le compilateur va chercher en deuxième élément de méthode un symbole représentant un champ. Or, il trouvera le signe "=". Il délivrera un message d'erreur, ne pouvant identifier la commande avec une expression qu'il sait traiter.

En revanche, si vous écrivez :

```
CHERCHER(LePtrTable->;LePtrChamp->=Vrai)
```

ou

```
CHERCHER([LaTable];LePtrChamp->=Vrai)
```

vous levez toute ambiguïté.

Constantes

Si vous créez vos propres ressources 4DK# (constantes), assurez-vous que les numériques soient de type Entier long (L) ou Réel (R) et les alphas de type Chaîne (S). Tout autre type génèrera un Warning.

5

Conseils d'optimisation

Il est difficile, voire impossible, de consigner une fois pour toutes une méthode pour “bien programmer”. Tout au plus pouvons-nous renouveler les recommandations émises dans la documentation de 4^e Dimension et les Notes Techniques et vous inviter à bien structurer vos programmes, grâce notamment aux possibilités de programmation générique offertes par 4^e Dimension.

De fait, il est clair que si vous essayez de compiler une base très bien structurée, vous obtiendrez de bien meilleurs résultats que si votre base est mal structurée. Par exemple, si vous écrivez une méthode projet générique qui gère n méthodes objet, vous obtiendrez de bien meilleurs résultats, en interprété comme en compilé, que si les n méthodes objet comportent n fois les mêmes séquences d'instructions.

La qualité de votre programmation peut donc avoir des effets sur la qualité du code compilé.

Vous améliorerez progressivement votre code 4^e Dimension grâce à l'habitude. De la même façon, c'est en utilisant fréquemment le compilateur que vous acquerrez les réflexes qui permettent inconsciemment d'aller droit au but.

En attendant cependant, voici quelques conseils et astuces qui vous permettront de gagner du temps dans des opérations simples et fréquentes.

Remarque préliminaire : les commentaires

Certaines astuces que nous vous conseillons peuvent rendre votre code moins lisible par une personne extérieure ou par vous-même ultérieurement. En conséquence, n'hésitez pas à assortir vos méthodes de commentaires détaillés. En effet, si les commentaires ont une tendance à ralentir une base interprétée, ils n'ont strictement aucune influence sur les temps d'exécution d'une base compilée.

Optimisation par les directives de compilation

Les directives de compilation peuvent vous aider à accélérer considérablement votre code.

Par défaut, le compilateur donne le type le plus large possible à vos variables afin de ne pas vous pénaliser. Si vous ne typerez pas par une directive de compilation une variable désignée par l'instruction : `Var:= 5` le compilateur lui donnera le type Réel, même s'il s'agit de l'affectation d'une valeur entière.

Nous allons maintenant voir comment, dans certains cas, déclarer une variable peut vous faire gagner beaucoup de temps.

Note Ces optimisations peuvent aussi se faire au niveau du projet de compilation, en sélectionnant les options de typage par défaut adéquates. Il conviendra néanmoins d'être prudent et de vérifier que les options de compilation sont bien identiques à chaque compilation de la base. Les directives de compilation sont, elles, toujours présentes dans la base.

Numériques

Par défaut, le compilateur donne le type Réel (numérique) aux variables numériques non typées par directive de compilation et si le projet de compilation n'indique pas le contraire. Or, les calculs sur un Réel sont plus lents que sur un Entier. Lorsque vous êtes sûr qu'un de vos numériques s'exprimera toujours dans votre base sous forme d'un Entier, il est avantageux de le déclarer par les directives de compilation `C_ENTIER` ou `C_ENTIER LONG`.

Une bonne habitude à prendre, par exemple, est de systématiquement déclarer vos compteurs de boucles comme Entier par directive de compilation.

Comparons les temps d'exécution d'une boucle vide de 1 à 5 000 000 :
Boucle(\$i;1;5000000)
Fin de boucle

Dans les deux cas suivants, sur Pentium 90 :

- \$i n'est pas déclaré par une directive de compilation (c'est-à-dire \$i de type Numérique) : 6 secondes
- \$i est déclaré par la directive de compilation C_ENTIER LONG : 0 seconde. La boucle est instantanée.

Note Ces durées peuvent varier selon la configuration de votre ordinateur.

Par ailleurs, certaines fonctions standard de 4^e Dimension renvoient des Entiers (exemple : Code ascii, Ent...). Si vous affectez le résultat d'une de ces fonctions à une variable non typée de votre base, 4D Compiler lui donnera le type Numérique et non Entier. Pensez donc à déclarer ces variables par des directives de compilation si vous êtes certain qu'elles ne seront utilisées que dans ces conditions.

Prenons un exemple simple. Une fonction renvoyant une valeur aléatoire comprise entre deux bornes peut s'écrire :

\$0:=Modulo(Hasard;(\$2-\$1+1))+\$1

Elle renvoie toujours une valeur entière. Si cette fonction est écrite ainsi, le compilateur donnera à \$0 le type Numérique et non le type Entier ou Entier long. Il est donc préférable d'écrire cette méthode sous la forme :

C_ENTIER LONG(\$0)

\$0:=Modulo(Hasard;(\$2-\$1+1))+\$1

Le paramètre rendu par la méthode sera plus petit et il sera donc beaucoup plus rapide de faire appel à cette méthode.

Prenons un autre exemple simple. Supposons que vous ayez déclaré deux variables en Entier long par l'instruction suivante :

C_ENTIER LONG(\$var1;\$var2)

et qu'une troisième variable non typée reçoive la somme des deux autres :

\$var3:=\$var1+\$var2.

Il vous faudra explicitement déclarer \$var3 comme Entier long si vous voulez effectivement que \$var3 soit de type Entier long, sinon 4D Compiler la typera par défaut en Numérique.

Note Attention au mode de calcul dans 4D. En mode compilé, ce n'est pas le type de la variable recevant le calcul qui détermine le mode de calcul, mais le type des opérandes.

- Dans l'exemple ci-dessous, le calcul s'effectue sur des Entiers longs :

```
C_REEL($var3)
C_ENTIER LONG($var1;$var2)
$var1:=2147483647
$var2:=1
$var3:=$var1+$var2
```

\$var3 prendra la valeur -2147483648 en mode compilé comme en interprété.

- Dans l'exemple suivant :

```
C_REEL($var3)
C_ENTIER LONG($var1)
$var1:=2147483647
$var3:=$var1+1
```

pour des raisons d'optimisation, 4D Compiler considère la valeur 1 comme une valeur entière. *\$var3* prendra alors la valeur -2147483648 en mode compilé (car le calcul s'effectue sur des Entiers longs) et 2147483648 en mode interprété (car le calcul s'effectue sur des Réels).

Les boutons sont un cas particulier de Numérique qu'il est possible de déclarer en Entier long.

Chaînes de caractères

De même que pour les numériques, le compilateur donne par défaut le type Texte à vos variables alphanumériques, si votre projet de compilation n'indique pas le contraire. Si vous écrivez :

```
MaChaine:="Bonjour"
```

MaChaine sera pour le compilateur une variable de type Texte.

Si vous avez beaucoup de traitements à effectuer sur cette variable, il est avantageux de la déclarer explicitement à l'aide de la directive C_ALPHA. Les traitements sont en effet beaucoup plus rapides sur les variables de type Alpha dont la longueur maximale est définie, que sur les variables de type Texte. N'oubliez pas cependant les règles de comportement de cette directive.

Si vous souhaitez tester la valeur d'un caractère, il est plus rapide de faire la comparaison sur son Code ascii que sur le caractère lui-même. En effet, la routine standard de comparaison de caractères prend en compte toutes les variations du caractère, comme par exemple les accentuations.

Remarques diverses

Tableaux à deux dimensions

Les traitements sur les tableaux à deux dimensions seront mieux gérés si la deuxième dimension est plus grande que la première.

Par exemple, un tableau déclaré sous la forme :

TABLEAU ENTIER(LeTableau;5;1000)

sera mieux géré qu'un tableau déclaré sous la forme :

TABLEAU ENTIER(LeTableau;1000;5)

Champs

Vous gagnerez du temps lorsque, si vous devez effectuer plusieurs calculs sur un champ, vous rangez la valeur de ce champ dans une variable et que vous effectuez vos calculs sur cette variable, plutôt que de faire directement les calculs sur le champ. Illustrons notre propos par un exemple.

Prenons la méthode suivante :

Au cas ou

```

: ([Client]Dest="Paris")
  Transport:="Coursier"
: ([Client]Dest="Corse")
  Transport:="Avion"
: ([Client]Dest="Etranger")
  Transport:="Service express"

```

Sinon

```

  Transport:="Poste"

```

Fin de cas

La séquence ci-dessus sera plus lente à exécuter que si elle avait été écrite de la façon suivante :

```

$Dest:=[Client]Dest

```

Au cas ou

```

: ($Dest="Paris")
  Transport:="Coursier"
: ($Dest="Corse")
  Transport:="Avion"
: ($Dest="Etranger")
  Transport:="Service express"

```

Sinon

```

  Transport:="Poste"

```

Fin de cas

Pointeurs

De même que pour les champs, il est plus rapide de travailler sur une variable intermédiaire que sur un pointeur dépointé. Vous gagnerez énormément de temps si, lorsque vous devez faire plusieurs calculs sur une variable pointée, vous rangez le pointeur dépointé dans une variable.

Vous disposez par exemple d'un pointeur pointant sur un champ ou sur une variable, `MonPtr`. Ensuite, vous souhaitez faire une série de tests sur la valeur pointée par `MonPtr`. Vous pouvez écrire :

Au cas ou

: (`MonPtr-> = 1`)

Séquence 1

: (`MonPtr-> = 2`)

Séquence 2

...

Fin de cas

Il est plus rapide d'exécuter cette série de tests si elle est écrite ainsi :

`Temp:=MonPtr->`

Au cas ou

: (`Temp = 1`)

Séquence 1

: (`Temp = 2`)

Séquence 2

...

Fin de cas

Variables locales

Utiliser des variables locales permet, dans bien des cas, de mieux structurer son code. Ces variables présentent d'autres avantages :

- Les variables locales prennent moins de place en mémoire lors de l'utilisation d'une base : en effet, elles sont créées lorsqu'on entre dans la méthode où elles sont utilisées et détruites dès qu'on sort de cette méthode.
- Le code généré est optimisé pour les variables locales, en particulier de type Entier long. Pensez-y pour vos compteurs de boucle.

6

Fenêtres des options

Ce chapitre décrit l'interface de 4D Compiler, c'est-à-dire la façon dont vous allez communiquer avec le programme : les menus et les fenêtres des options.

Nous décrirons ensuite en détail le processus de compilation.

Remarques préliminaires

Une base à compiler peut être ouverte à la fois par 4^e Dimension et par le compilateur.

Ceci vous évite de quitter et de rouvrir une base à chaque fois que vous souhaitez la compiler. Il est simplement nécessaire, le cas échéant, de fermer le fichier d'erreurs en sélectionnant la commande **Arrêter la correction** du menu **Mode** de 4^e Dimension.

Durant la compilation, vous ne pouvez pas agir sur la structure de votre base. Une fois la compilation terminée, vous pourrez retourner à votre base sans avoir besoin de relancer 4^e Dimension.

Les menus de 4D Compiler

Double-cliquez sur l'icône de 4D Compiler pour entrer dans le programme. Les menus suivants vous sont proposés :

- **Fichier**
- **Edition**
- **Aide** (Windows uniquement)

La présence d'un menu **Edition** est obligatoire dans toute application. Il n'est d'aucune utilité pour 4D Compiler.

Le seul menu qui vous importe donc dans 4D Compiler est le menu **Fichier**. Il comporte neuf commandes :



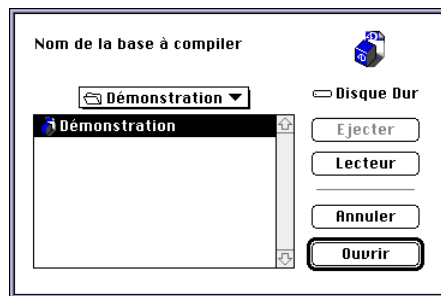
Nouveau...

Cette commande vous permet de demander la compilation d'une nouvelle base. Lorsque vous avez choisi cette commande, le compilateur affiche la boîte de dialogue standard d'ouverture de fichiers. Localisez la base que vous voulez compiler.

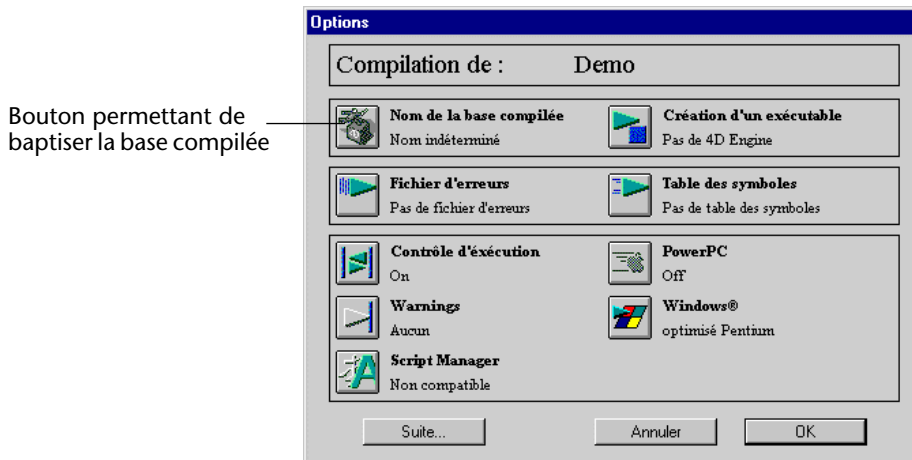
Windows



MacOS



Dès que vous indiquez votre choix, la fenêtre d'options apparaît :

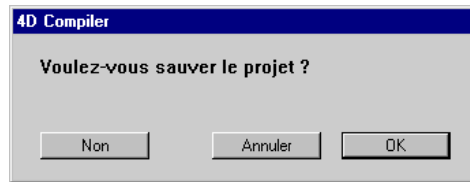


Ouvrir...

Lorsque vous demandez **Nouveau**, vous vous apprêtez à procéder à une nouvelle compilation. Lorsque vous choisissez la commande **Ouvrir**, vous pouvez ouvrir un projet de compilation déjà existant.

La notion de projet

Comme nous l'avons vu dans l'exemple d'initiation, vous choisissez vos options pour une compilation. Vous avez pu noter que, après le choix des options, qui dans l'exemple se limitait à une seule, 4D Compiler a affiché la boîte de dialogue suivante :



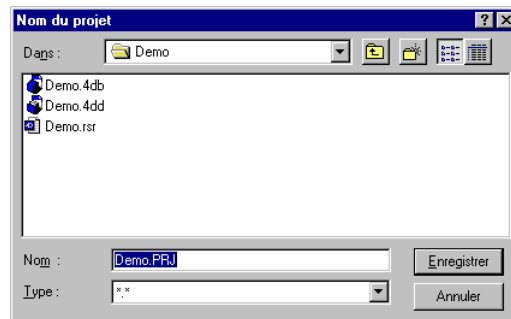
Lorsque vous cliquez sur **OK**, vous pouvez enregistrer votre environnement.

L'intérêt de sauvegarder cet environnement consiste à pouvoir effectuer rapidement plusieurs compilations d'une même base puisqu'on a conservé intégralement l'environnement de compilation.

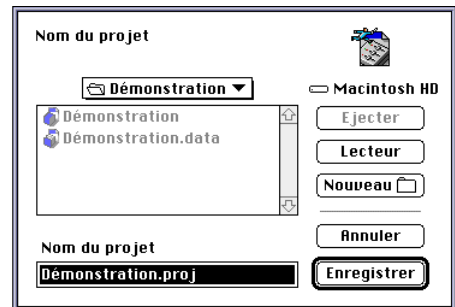
Vous pouvez en effet enregistrer tous les paramètres d'une compilation : c'est ce qu'on appelle un projet.

Une boîte de dialogue d'enregistrement de fichiers vous est présentée :

Windows



MacOS



Par défaut, le nom du projet est :

- sous Windows, MaBase.prj,
- sous MacOS, MaBase.proj.

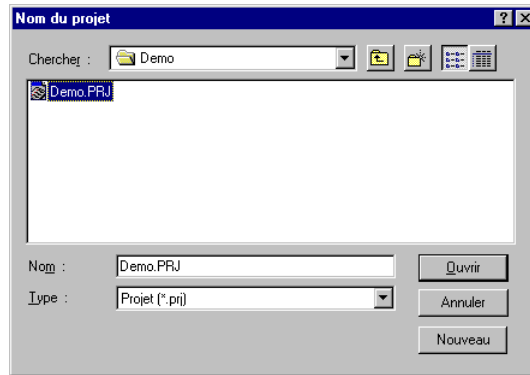


Baptisez votre projet et cliquez sur **Enregistrer**. Un fichier projet est représenté au niveau du bureau par l'icône ci-contre.

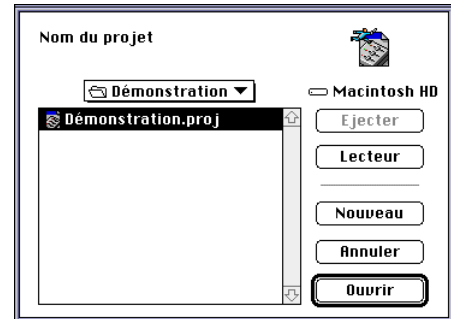
Ouvrir un projet

La commande **Ouvrir** permet d'ouvrir un projet déjà existant. Dès que vous choisissez cette commande, une boîte de dialogue classique d'ouverture de document vous est présentée.

Windows



MacOS



Vous notez la présence du nom de votre projet dans cette boîte de dialogue. Sélectionnez-le et cliquez sur le bouton **Ouvrir**.

4D Compiler affiche la fenêtre d'options du projet enregistré.



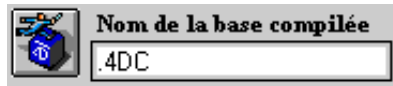
Si vous avez demandé **Ouvrir** alors que vous n'avez en fait pas de projet existant, le bouton **Nouveau** vous permet de vous replacer dans les mêmes conditions que si vous aviez demandé directement **Nouveau** dans le menu **Fichier**.

Recompiler	<p>Cette commande permet de relancer la compilation d'un projet sans repasser par la boîte de dialogue des options.</p> <p>Que vous relanciez une compilation par la commande Recompiler ou que vous ouvriez un projet déjà existant, 4D Compiler prend en compte l'ensemble des options que vous avez définies.</p> <p>Comme vous le verrez au paragraphe suivant qui détaille l'ensemble des options, certaines d'entre elles génèrent des fichiers dont vous choisissez les noms. La relance d'un projet génère à nouveau et automatiquement tous les fichiers correspondants.</p> <p>Faites attention par conséquent à ne pas écraser des fichiers que vous souhaiteriez conserver. Nous reviendrons sur ce point au moment où nous décrirons les fichiers concernés.</p> <hr/> <p><i>Note</i> Lorsque vous indiquez la base à compiler, soit en ouvrant un projet déjà existant, soit en créant un nouveau projet, 4D Compiler vous réclame au début de la compilation le mot de passe Super_Utilisateur, si la base est protégée par un mot de passe.</p> <hr/>
Fermer	<p>Cette commande permet de fermer la fenêtre de travail du compilateur à la fin de la compilation.</p>
Enregistrer	<p>Cette commande, toujours disponible, provoque l'enregistrement du projet courant à la place de la version précédente.</p>
Enregistrer sous...	<p>Cette commande affiche la boîte de dialogue standard de création de fichiers dans lequel vous pourrez spécifier un nouveau nom pour le projet courant.</p>
Version précédente	<p>Cette commande vous permet de revenir à la dernière version sauvegardée du projet courant.</p>
Projet par défaut	<p>Cette commande vous permet de définir des options qui, par défaut, seront appliquées à chaque nouveau projet.</p> <ul style="list-style-type: none">■ Sélectionnez la commande Projet par défaut... La fenêtre d'options apparaît.

Le projet que vous définirez ainsi sera sauvegardé dans un fichier de préférences. Les options que vous aurez choisies dans le projet par défaut seront reportées dans chaque nouveau projet, quelle que soit la base.

Le choix des options s'effectue de façon identique, que l'on définisse un projet par défaut ou un projet particulier à une base. La seule différence est que les boîtes de dialogue de création de fichiers sont remplacés par une zone dans laquelle vous inscrirez un suffixe qui sera celui utilisé pour les bases qui se serviront de ce projet par défaut.

- ▼ Exemple : cliquez sur le bouton **Nom de la base compilée**. Une zone éditable apparaît :



Saisissez le suffixe voulu, par exemple "CP".

Par défaut, le nom de la prochaine base compilée sera le nom de la base interprétée suivi du suffixe que vous venez d'indiquer.

Ce projet par défaut peut être modifié au cas par cas : par exemple, si en général, vous avez besoin d'une compatibilité Script Manager sous MacOS et que pour une base particulière, cette option n'est pas nécessaire, vous pouvez opérer la modification à la volée. Cette modification sera prise en compte pour le projet sur lequel vous travaillez au moment de la modification, mais ne sera pas enregistrée dans le projet par défaut lui-même.

Lorsque vous voulez modifier un projet par défaut, appelez ce projet en sélectionnant **Projet par défaut...** Modifiez votre projet et cliquez sur le bouton **Sauver**.

Quitter

Cette commande vous permet de quitter 4D Compiler.

Fonctionnalités d'interface

Drag and drop (glisser-déposer)

Il est possible de lancer 4D Compiler par "drag and drop" (opération qui consiste à faire glisser puis à déposer l'icône du fichier à ouvrir sur celle de l'application).

Glisser-déposer d'un fichier structure

Vous pouvez effectuer un drag and drop du fichier structure d'une base 4D sur l'icône de 4D Compiler. Vous obtenez alors la fenêtre de définition de projet.

Si vous réalisez la même opération en maintenant la touche **Alt** (sous Windows) ou **Option** (sous MacOS) enfoncée, la compilation débutera immédiatement, suivant les paramètres du projet par défaut, si vous en avez défini un.

Glisser-déposer d'un fichier projet

Vous pouvez effectuer un drag and drop d'un fichier projet sur l'icône de 4D Compiler. Vous obtenez alors la fenêtre de définition de ce projet.

Si vous réalisez la même opération en maintenant la touche **Alt** (sous Windows) ou **Option** (sous MacOS) enfoncée, la compilation débutera immédiatement, suivant les paramètres du projet ouvert.

Les fenêtres d'options pour la compilation

La première fenêtre propose des options générales. La seconde fenêtre à laquelle vous accédez en cliquant sur le bouton **Suite...** vous permet des réglages plus fins de votre projet.

Première fenêtre d'options

Cette fenêtre comporte dix icônes.



Chacune de ces icônes présente deux ou trois visages. Cliquer dessus vous permet de modifier son état.

Nom de la base compilée

Icône non sélectionnée



Icône sélectionnée

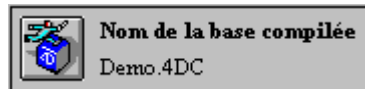


Lorsque vous cliquez sur l'icône, la boîte de dialogue standard de création de documents s'ouvre.

Par défaut, le nom donné à la base compilée est le nom de la base originale suivi du suffixe .comp sous MacOS et de .4DC sous Windows ou, s'il existe, du suffixe indiqué dans le projet par défaut.

Vous pouvez modifier ce nom à votre convenance. Cliquez sur le bouton **Enregistrer**.

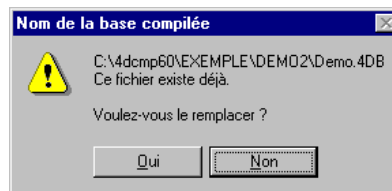
Vous revenez à la fenêtre des options : le nom de la base compilée est inscrit à droite de l'icône.



Si la base originale et la base compilée se trouvent au même endroit, elles doivent porter deux noms différents.

Si vous essayez de donner à la base compilée le même nom que la base originale, la boîte de dialogue d'alerte standard de remplacement de document du système d'exploitation apparaîtra d'abord :

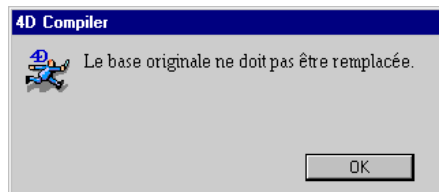
Windows



MacOS



Même si vous cliquez sur **Oui**, 4D Compiler vous interdira cette action et vous délivrera un message :



Une base compilée est une image de la base originale et fonctionne exactement comme elle, à ceci près que dans la base compilée, vous n'avez plus accès à la structure. On comprend donc que 4D Compiler vous interdise absolument de faire une action irréversible ou irrémédiable.

Création d'un exécutable

Icône non sélectionnée



Icône sélectionnée



■ Windows

Cette option vous permet de créer une application 4^e Dimension directement exécutable.

Avant de lancer la compilation, assurez-vous d'avoir installé les fichiers suivants sur votre disque dur, et que ceux-ci se trouvent dans le même répertoire :

4DENGINE.4DE

4DENGINE.RSR

ASIFONT.FON

ASIPORT.RSR

ASINTPPC.DLL

Les fichiers ASIFONT.FON, ASIFONT.MAP et 4DMSG.DLL sont facultatifs.

1 Dans la fenêtre d'options, cliquez sur l'icône "Création d'un exécutable".

4D Compiler affiche alors la boîte de dialogue d'ouverture de documents.

2 Sélectionnez le fichier "4DEngine.4DE".

3 Cliquez sur le bouton OK.

Lorsque vous revenez à la fenêtre d'options, votre choix est confirmé par la mention du 4D Engine sélectionné.

4 Donnez un nom à votre exécutable dans la zone "Nom de la base compilée". Choisissez le répertoire dans lequel vous voulez l'enregistrer.

Un message vous annonçant la création du répertoire DISTRIB est affiché au début de la compilation.

A la fin de la compilation, vous devez trouver dans le répertoire de destination, DISTRIB, les fichiers suivants :

- NomDeLaBase.EXE qui est votre exécutable,
- NomDeLaBase.4DC et NomDeLaBase.RSR qui composent votre structure compilée.

Note Vous ne pourrez pas ouvrir les fichiers cités ci-dessus avec 4^e Dimension, 4D Server ou 4D First.

- ASIFONT.FON
- ASINTPPC.DLL
- ASIPORT.RSR

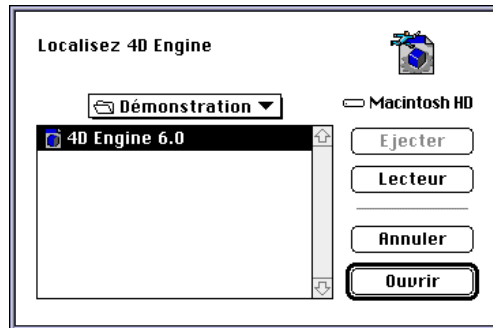
Il est à noter que tous ces fichiers sont indissociables.

- ▶ Pour achever la création de votre exécutable, vous devez déplacer manuellement dans votre répertoire de destination les fichiers complémentaires suivants :
 - le fichier de données,
 - le répertoire WIN4DX et le répertoire MAC4DX, si vous possédez des Plug-ins.
- ▶ Pour la distribution de votre application, utilisez un installeur (par exemple 4D InstallMaker de 4D). L'installeur vous permettra :
 - de proposer à vos utilisateurs une installation standard ou personnalisée de votre application ;
 - de préparer une installation soignée sur plusieurs disquettes ;
 - de personnaliser vos icônes au niveau du Gestionnaire de programmes.

■ MacOS

L'option **Création d'un exécutable** vous permet de fusionner votre base avec le fichier 4D Engine, et ainsi, créer une application double-cliquable.

- 1 **Avant de lancer le compilateur, assurez-vous d'avoir installé le fichier 4D Engine sur votre disque dur.**
- 2 **Dans la fenêtre d'options, cliquez sur l'icône correspondante.**
4D Compiler affiche alors une boîte de dialogue d'ouverture de documents.
- 3 **Sélectionnez le fichier 4D Engine.**



- 4 **Cliquez sur le bouton Ouvrir.**

Lorsque vous revenez à la fenêtre d'options, le choix d'une fusion avec le fichier 4D Engine est confirmé par la mention de 4D Engine. Le compilateur regroupe pendant la compilation le fichier 4D Engine et la structure compilée de la base.

Note Vous trouverez à l'[annexe A, "Personnaliser son application", page 135](#) la marche à suivre pour personnaliser l'icône de la base.

Fichier d'erreurs

Icône non sélectionnée



Icône sélectionnée

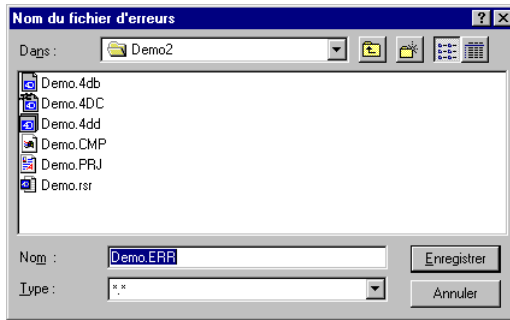


Si vous sélectionnez cette option, le compilateur générera, s'il y a lieu, un fichier compatible 4^e Dimension contenant la liste des erreurs détectées au typage et à la compilation.

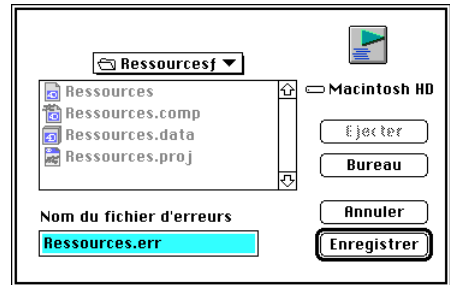
Ce fichier contiendra aussi le cas échéant les Warnings de compilation (voir le paragraphe ["Warnings", page 94](#)).

Lorsque vous cliquez sur l'icône, 4D Compiler affiche une boîte de dialogue standard de création de documents. Par défaut, il donne à ce fichier le nom `NomDeLaBase.err`.

Windows



MacOS



Vous pouvez modifier ce nom à votre convenance. Cliquez sur le bouton **Enregistrer**.

Le nom choisi pour le fichier d'erreurs est désormais inscrit dans la fenêtre des options.

Nous expliquons dans le [chapitre “Les aides à la compilation”, page 109](#), l'intérêt pratique et le mode d'utilisation de ce document.

Vous pouvez utiliser ce fichier de deux manières :

- soit comme fichier Texte, vous permettant de garder une trace écrite des messages délivrés par le compilateur pour la base compilée ;
- soit à l'intérieur de 4^e Dimension, ce qui vous permet de corriger vos bases en temps réel, s'il y a lieu.

Table des symboles

Icône non sélectionnée



Icône sélectionnée

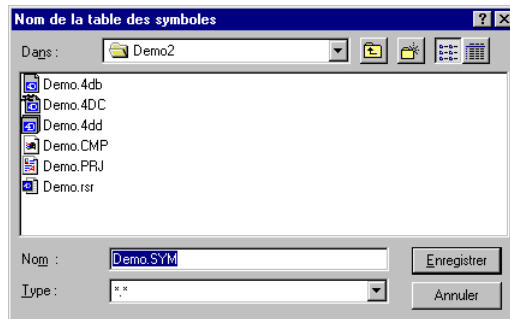


Si vous sélectionnez cette option, le compilateur générera un fichier de type ASCII (texte seul) qui sera la représentation de la table des symboles de votre base.

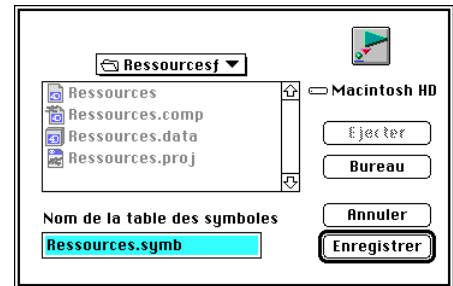
Ce fichier contiendra la liste de vos variables accompagnées de leur type et du nom de la méthode à partir de laquelle le type a été déduit. La table des symboles contient également la liste de vos méthodes et fonctions accompagnées du type de leurs paramètres et du type d'un résultat éventuel.

Lorsque vous cliquez sur l'icône, 4D Compiler affiche une boîte de dialogue standard de création de documents.

Windows



MacOS



Par défaut, le fichier est nommé :

- sous Windows, NomDeLaBase.sym,
- sous MacOS, NomDeLaBase.symb.

Vous pouvez modifier ce nom à votre convenance. Cliquez sur le bouton **Enregistrer**. Le nom choisi pour la table des symboles est désormais inscrit dans la fenêtre des options.

Pour plus d'informations sur l'intérêt pratique et le mode d'utilisation de ce document, reportez-vous au [chapitre "Les aides à la compilation"](#), page 109.

Contrôle d'exécution

Icône non sélectionnée



Icône sélectionnée



Lorsque vous sélectionnez cette option, vous n'obtenez aucun message en direct, sous une forme ou sous une autre de la part du compilateur lors du processus de compilation. Vous ne pourrez percevoir les effets de cette sélection que lors de l'utilisation de la base compilée. En effet, il peut exister certaines particularités dans votre code qui ne sont détectables qu'à l'exécution de la base.

Cette option vous permet un niveau de contrôle sur les bases particulièrement puissant. Nous reviendrons sur cette notion dans le [chapitre "Les aides à la compilation"](#), page 109.

Note Cette option peut ralentir sensiblement l'exécution du code compilé.

Script Manager

Icône non sélectionnée
Non compatible



Icône sélectionnée
Compatible



Choisissez cette option si votre base doit être utilisée avec une version de 4^e Dimension utilisant le Script Manager sous MacOS ou les alphabets non romans sous Windows.

Le Script Manager est une possibilité offerte par les systèmes d'exploitation, permettant aux logiciels qui sont conçus pour l'utiliser de fonctionner avec des caractères non romans, par exemple en japonais, en chinois, en arabe ou en hébreu. Si vous disposez d'une version de 4^e Dimension fonctionnant sous Script Manager, sélectionnez l'icône.

Dans tous les autres cas, évitez de sélectionner cette option : vous ralentiriez inutilement le code généré.

Warnings

Icône Warnings Off



Icône Warnings On



Icône Warnings Plus



Cette option est un outil particulièrement puissant qui vous permet d'aller plus loin dans la vérification de vos bases.

Elle est précieuse dans deux cas :

- vous n'avez pas été tout à fait assez soigneux dans votre code mais il reste compilable. Vous voulez néanmoins vous assurer d'une qualité supérieure pour votre code.
- vous utilisez des instructions sur lesquelles le compilateur ne peut pas avoir d'opinion.
A ce moment-là, le compilateur fait une déduction logique d'après ce que vous avez écrit mais vous voulez vérifier que cette déduction correspond à votre intention. Si cette option est sélectionnée, le compilateur vous délivre des messages d'information et non des messages d'erreur. Ces messages seront automatiquement sauvegardés dans le fichier d'erreurs.

Lorsque cette option n'est pas sélectionnée, le compilateur ne vous délivre aucun warning.

Si l'option est sur la position "On", le compilateur vous délivre durant la compilation les Warnings simples.

Si l'option sélectionnée est "Plus", le compilateur vous délivre les Warnings simples ainsi que les Warnings Plus.

Note Nous revenons sur les possibilités offertes par cette option dans le [chapitre "Les aides à la compilation", page 109](#)).

Choix du code

Icône pour code PowerPC



Non sélectionnée



Sélectionnée

Note Un seul choix est présenté pour le code PowerPC, car le même algorithme d'optimisation améliore le code pour toute la gamme de processeurs PowerPC.

Icône pour code 80386/486



Icône pour code optimisé Pentium



Le compilateur vous permet de choisir l'assembleur pour lequel vous voulez compiler.

Cet assembleur dépend du microprocesseur de la machine sur lequel votre base de données ou votre application 4^e Dimension va être utilisée et non de la machine sur laquelle vous compilez.

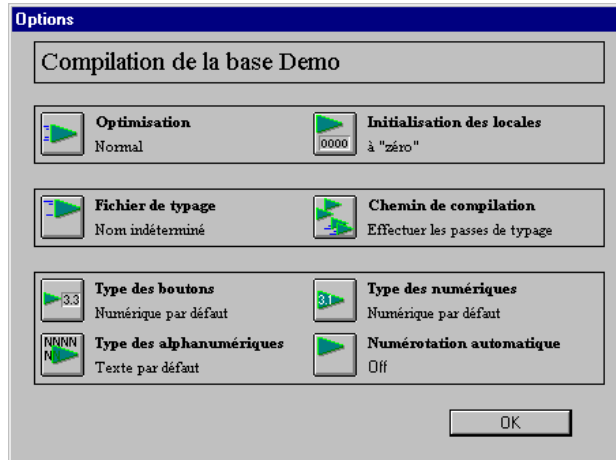
Le Pentium est plus puissant que les processeurs 80386/486. Vous pouvez par cette option de compilation en tirer parti.

Si vous ne savez pas quel microprocesseur équipe votre ordinateur, reportez-vous au manuel qui vous a été fourni avec votre machine.

Note Une base compilée pour Pentium fonctionnera sur les machines 80386/486.

Deuxième fenêtre d'options

Vous accédez à la deuxième fenêtre d'options en cliquant sur le bouton **Suite...** situé au bas de la première fenêtre d'options. Une seconde fenêtre similaire à la première est alors affichée. Elle contient huit options.



Nous allons examiner chacune de ces options en détail.

Optimisation

Icône non sélectionnée
Normal



Icône sélectionnée
Optimisé



4D Compiler vous donne le choix entre deux générateurs de code :

- le premier vous permet une compilation rapide et la génération d'un bon code de travail.
- le second génère un code plus optimisé et plus compact. Le processus de génération du code sera plus lent, mais à l'exécution votre base sera plus rapide. Nous vous recommandons d'utiliser cette option en phase finale de vos développements.

Initialisation des locales

A zéro



A une valeur aberrante



Non



Cette option vous propose trois choix. Elle intervient dans la base compilée, au moment de l'appel d'une méthode.

Lorsque vous entrez dans une méthode, la place pour les variables locales est réservée. Ces variables locales seront détruites en sortie de méthode.

A l'entrée dans la méthode, vous pouvez demander au code compilé d'adopter l'un des trois comportements suivants :

- créer les variables locales et les initialiser à la valeur nulle par défaut (chaîne vide pour les chaînes de caractères, 0 pour les numériques...). C'est cette option que vous choisissez lorsque vous n'avez pas initialisé vos variables.
- créer les variables et ne pas les initialiser par défaut. L'entrée dans la méthode se fera plus rapidement, puisque le compilateur n'a pas à faire les initialisations. Vous gagnez ainsi du temps lors de l'exécution de la base, à condition que vos initialisations soient correctes.
- créer les variables et les initialiser par défaut à une valeur aberrante. Cette option vous permet de repérer très précisément quelles sont les locales que vous avez oublié d'initialiser. 4D Compiler vous retournera une valeur aberrante, toujours la même, (1919382119 pour les entiers longs, "rgrg" pour les chaînes de caractères, Vrai pour les booléens...) pour attirer votre attention.

Lors de la compilation pour une version finale d'un produit, vous choisirez la première ou la deuxième option.

Fichier de typage

Icône non sélectionnée



Icône sélectionnée



A votre demande, 4D Compiler peut générer un fichier contenant les directives de compilation déclarant toutes vos variables. Ensuite, si vous souhaitez reporter ces déclarations dans la structure de votre base 4^e Dimension, vous collez le document là où vous en avez besoin, dans une ou plusieurs méthodes. Reportez-vous au paragraphe ["Où placer vos directives de compilation ?"](#), page 33.

Chemin de compilation

Effectuer les passes de typage



Les variables globales sont typées



Toutes les variables sont typées



Ces trois positions ont les actions suivantes :

- Effectuer les passes de typage
Dans ce cas, 4D Compiler passe par toutes les étapes qui permettent la compilation.
- Les variables globales sont typées
Cette option se justifie lorsque vous avez assuré le typage de toutes vos variables process et interprocess, soit vous-même, soit en utilisant le fichier de typage (option décrite ci-dessus). 4D Compiler accélère la compilation en n'effectuant qu'une seule passe au lieu de quatre.

La compilation suivra ainsi le chemin suivant :

Copie de la base —> Passe de compilation —> Eventuellement, génération des fichiers demandés.

Vous faites l'économie des trois passes de typage.

- Toutes les variables sont typées
Utilisez cette option lorsque toutes vos variables process, interprocess et locales sont typées, sans ambiguïté. La compilation sera beaucoup plus rapide. Cette option est une option de travail vous permettant de finaliser vos bases plus rapidement mais n'a aucun impact sur le code généré.

Note Cette option est à utiliser avec précaution car elle implique une diminution du nombre des warnings envoyés par le compilateur. D'autre part, les éventuelles erreurs de typage ne seront pas repérées.

Type par défaut des boutons

Entier long par défaut



Numérique par défaut



Par défaut, le compilateur donne toujours le type le plus large possible aux variables non identifiées. Par cette option, vous pouvez forcer le typage des boutons d'une manière univoque, soit en réel, soit en entier long.

Cette option n'a pas de priorité sur les directives qui auraient été éventuellement placées dans votre base. Elle concerne aussi les variables suivantes :

- cases à cocher
- cases à cocher 3D
- boutons
- boutons inversés
- boutons invisibles
- boutons 3D
- boutons image
- grille de boutons
- boutons radio
- boutons radio 3D
- radio image
- menus image
- menus déroulants hiérarchiques
- listes hiérarchiques

Cette option vous permet, si vous choisissez le type Entier long, d'optimiser vos bases au niveau de leur exécution.

Type par défaut des numériques

Numérique par défaut



Entier long par défaut



Par défaut, le compilateur donne toujours le type le plus large possible aux variables non identifiées. Par cette option, vous pouvez forcer le typage des numériques d'une manière univoque, soit en réel, soit en entier long. Cette option n'a pas de priorité sur les directives qui aurait été éventuellement placées dans votre base.

Cette option vous permet d'optimiser vos bases au niveau de l'exécution, si vous choisissez le type Entier long.

Type par défaut des alphanumériques

Texte par défaut



Chaîne fixe



Par défaut, le compilateur donne toujours le type le plus large possible aux variables non identifiées. Par cette option, vous pouvez forcer le typage des chaînes de caractères d'une manière univoque, soit en texte, soit en chaîne fixe. Cette option n'a pas de priorité sur les directives qui auraient été éventuellement placées dans votre base.

Si vous choisissez de donner par défaut à vos chaînes de caractères le type chaîne fixe, une zone saisissable apparaît et vous permet d'indiquer au compilateur la longueur de ces chaînes par défaut.

Cette option vous permet d'optimiser vos bases au niveau de l'exécution, si vous choisissez le type chaîne fixe.

Note Le compilateur type les variables suivant trois critères, classés ci-dessous par ordre de priorité :

- vos directives de compilation qui sont toujours prioritaires.
- les options de typage par défaut.
- aux variables non concernées par le typage par défaut et non déclarées, le compilateur essaiera de donner le type adéquat.

Numérotation automatique

Icône non sélectionnée



Icône sélectionnée



Si vous sélectionnez cette option, 4D Compiler donnera automatiquement à la base compilée un numéro de version égal à 1.0dx.

Note La lettre *d* signifie *développement*.

Ce numéro de version est enregistré dans le projet et *x* est incrémenté automatiquement à chaque nouvelle compilation.

Lorsque vous sélectionnez cette option, une zone saisissable vous permet de saisir ou de modifier la valeur de x à partir de laquelle le compilateur va numéroté vos versions.

■ **MacOS**

Le compilateur crée, si elle n'existe pas, une ressource de type "vers", utilisée par la commande **Lire les informations** du Finder du Macintosh. S'il y en a déjà une dans votre structure originale, 4D Compiler la modifiera.

■ **Windows et MacOS**

Vous pouvez visualiser le numéro de version de votre application compilée grâce à la commande **APPELER SUR A PROPOS** qui, en plus des informations concernant la version de 4^e Dimension et de 4D Compiler, vous indiquera le numéro de version de votre application compilée.

Remarques générales sur les options

Aucune des options de compilation n'est obligatoire. Vous pouvez même lancer une compilation sans choisir aucune option. Dans ce cas, lorsque vous cliquez sur le bouton **OK**, le compilateur vous demande sous quel nom vous souhaitez baptiser la base compilée.

La boîte de dialogue d'enregistrement de documents s'affiche.

Note Vous pouvez parfaitement définir un projet sans préciser de nom pour la base à compiler. Simplement, lorsque vous lancerez la compilation 4D Compiler vous demandera de baptiser la base compilée.

Lancement d'une compilation

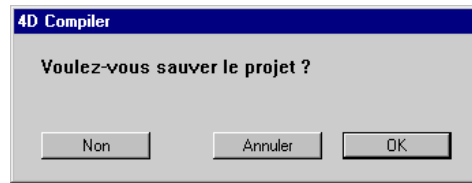
Lorsque vous avez choisi vos options de compilation et éventuellement enregistré votre environnement, lancez la compilation en cliquant sur le bouton **OK** de la fenêtre d'options.

Le compilateur travaille alors tout seul sans votre intervention.

Il s'interrompt seulement dans le cas où il n'arrive pas à localiser vos routines externes.

Nous détaillons ce point dans le paragraphe concernant le processus de compilation.

Si vous cliquez sur le bouton **OK**, sans avoir auparavant enregistré le projet courant, 4D Compiler vous propose de le faire par l'intermédiaire de la boîte de dialogue suivante :



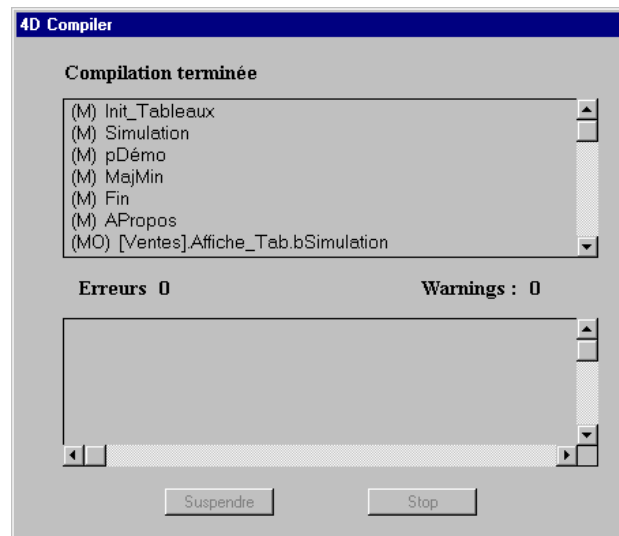
Le processus de compilation

Remarque préliminaire

Avant de compiler votre base, assurez-vous qu'elle fonctionne correctement déjà sous 4^e Dimension, non compilée. En effet, une base qui ne fonctionne pas sous 4^e Dimension ne fonctionnera pas mieux compilée.

Lorsque vous avez cliqué sur le bouton **OK**, le compilateur travaille tout seul.

La fenêtre de travail du compilateur est la suivante :



Dans la partie supérieure de la fenêtre, vous voyez défiler les méthodes bases, les méthodes projet, les triggers, les méthodes formulaire et les méthodes objet au fur et à mesure qu'elles sont traitées par le compilateur.

Dans la partie inférieure de la fenêtre, 4D Compiler délivre la liste des messages et des erreurs éventuelles. Cette fenêtre est surmontée de deux compteurs : le compteur des erreurs et celui des warnings, automatiquement incrémentés au fur et à mesure du processus de compilation.

Les deux boutons, **Suspendre** et **Stop**, situés en bas de cette fenêtre, permettent d'interrompre provisoirement (Suspendre) ou définitivement (Stop) la compilation. Nous reviendrons sur l'interruption du processus de compilation au paragraphe suivant.

Les principales étapes du processus sont les suivantes :

- copie de la base,
- typage des variables,
- compilation proprement dite.
- génération d'un exécutable.

Une petite icône se déplaçant sur la gauche de la fenêtre supérieure vous permet de savoir à quel stade se trouve la compilation.

Note Vous contrôlez le chemin de compilation grâce à une option de la deuxième fenêtre. Pour plus de détails, reportez-vous au ["Chemin de compilation", page 97](#).

Copie de la base

Seul le fichier structure de la base est dupliqué puis compilé.

Ce fichier contient entre autres les méthodes de votre base.

Les données, indépendantes de la structure, pourront indifféremment être exploitées par la base compilée et la base non compilée.

Note La structure originale n'est jamais modifiée par 4D Compiler.

Typage des variables

Durant cette phase, le compilateur crée la table des symboles de la base compilée. Cette phase se compose de trois passes :

- la passe de repérage des directives de compilation et des déclarations de tableau : cette passe est appelée passe de typage directif.
- la passe d'identification de type des variables process et interprocess : cette passe est appelée typage des variables process et interprocess.

- la passe d'identification de type des variables locales : cette passe est appelée typage des locales.

Chacune de ces passes est illustrée par une icône. Si vos méthodes sont petites et/ou votre machine rapide, vous les verrez à peine.

La passe de typage directif

Cette passe est représentée par l'icône suivante :



Lors de cette passe, le compilateur retrouve et stocke les types des variables faisant l'objet d'une directive de compilation. Il repère de même les tableaux de la base et leur type.

La passe de typage des variables process et interprocess

Cette passe est représentée par l'icône suivante :



Durant cette passe, le compilateur retrouve et stocke les types des variables process et interprocess simples qui n'ont pas fait l'objet d'une directive de compilation. Il procède à leur typage en fonction de leur utilisation.

La passe de typage des variables locales

Cette passe est représentée par l'icône suivante :



Lors de cette passe, le compilateur retrouve et stocke les types des variables locales faisant l'objet d'une directive de compilation. Il repère de même les tableaux locaux.

Remarque générale sur cette phase de typage

C'est dans cette phase que sont révélés les éventuels conflits de type que nous avons signalés dans le guide du typage.

En cas de conflit de type, 4D Compiler privilégie la première occurrence de la variable, à l'intérieur de chaque passe.

Les méthodes projet et les méthodes base sont examinées dans l'ordre où elles ont été créées dans 4^e Dimension.

Ensuite sont examinés successivement les triggers, les méthodes formulaire et les méthodes objet associées au formulaire en cours d'examen, dans l'ordre de création des tables et des formulaires.

Conclusion de la phase de typage

Lorsque le typage s'est bien passé ou, en tout cas, que les erreurs détectées n'empêchent pas de passer en passe de compilation, 4D Compiler passe automatiquement en passe de compilation, décrite plus bas.

Dans le cas contraire, c'est-à-dire lorsqu'il a détecté des erreurs générales de typage, 4D Compiler ne passe pas à la compilation mais effectue une passe supplémentaire.

Cette passe est illustrée par l'icône suivante :



Ces erreurs dites erreurs générales de typage ont deux causes :

- le compilateur a rencontré deux objets portant le même nom : une méthode et une variable, une variable et une commande de Plug-in, une commande de Plug-in et une variable, deux méthodes projet, etc.

Si 4D Compiler a rencontré deux variables portant le même nom, il signale le conflit de type mais ne s'arrête pas à ce niveau.

- Le compilateur a rencontré une variable process ou interprocess pour laquelle il est dans une impossibilité absolue de faire une estimation de type.

Il vous appartient alors, à l'aide des messages délivrés par le compilateur, de les corriger.

Compilation

La compilation proprement dite est la passe durant laquelle 4D Compiler va traduire et sauvegarder vos méthodes en assembleur. Durant cette phase, le compilateur retrouvera également le type des variables locales, sauf si vous avez spécifié le contraire par l'option **Chemin de compilation**.

Dans cette passe, 4D Compiler continue à rechercher des erreurs. Ces erreurs ne sont plus des erreurs de typage mais des erreurs de compilation (syntaxe, incohérences diverses...).

Cette passe est illustrée par l'icône suivante :



Après la passe de compilation, si le compilateur n'a détecté aucune erreur, que ce soit durant le typage ou la compilation, la base compilée est créée.

Interruption du processus de compilation

Si le compilateur a détecté des erreurs de typage ou des erreurs de compilation, la base compilée ne peut donc être créée et vous devez procéder aux corrections.

En cours de typage ou de compilation, vous pouvez à tout moment interrompre le processus en cliquant sur les boutons **Suspendre** ou **Stop**.

Ces interruptions sont utiles dans le cas où le compilateur vous a délivré des erreurs ou des warnings.

En effet, vous pouvez naviguer d'une méthode à une autre lorsque des messages sont attachés à cette méthode. Cette navigation se fait de deux façons :

- soit avec la souris. Déplacez le curseur de défilement et cliquez sur la méthode qui vous intéresse.
- soit avec la touche **Tab** pour vous déplacer vers le bas, et la combinaison des touches **Maj+Tab** pour vous déplacer vers le haut.

Le compilateur vous indique qu'un message est attaché à une méthode de deux façons, suivant la nature du message :

- lorsqu'il détecte une erreur dans une méthode, le compilateur met le nom de cette méthode en gras.
- lorsqu'il détecte un warning dans une méthode, le compilateur met le nom en italique.

Sélectionnez alors la méthode :



La nature de l'erreur est immédiatement détaillée dans la fenêtre inférieure. Ces trois lignes sont :

- pour la première, le numéro de ligne dans la méthode.
- pour la seconde, la ligne concernée par l'erreur ou le warning.
- pour la troisième, un diagnostic sur la nature de l'erreur.

Génération d'un exécutable

Si vous avez choisi l'option **Création d'un exécutable**, le compilateur dupliquera également le fichier 4D Engine durant cette phase.

Utilisation d'une base compilée

Lancez 4^e Dimension ou 4D Server et ouvrez la base compilée de la même façon que vous ouvrez toute base 4^e Dimension. Simplement, le mode **Structure** est toujours désactivé.

Si vous avez demandé la création d'un exécutable de votre base compilée, double-cliquez sur l'application obtenue.

Par défaut, l'icône de l'application se présente comme suit :



Si vous souhaitez personnaliser votre icône, reportez-vous à l'[annexe A, "Personnaliser son application", page 135](#).

Le fonctionnement de votre base compilée sera le même que celui de votre base non compilée.

7

Les aides à la compilation

Les aides à l'analyse et à la correction des bases sont de quatre types :

- l'aide à l'analyse proprement dite est fournie par la table des symboles. Cette table des symboles vous permettra de vous repérer plus rapidement dans vos variables. C'est un outil précieux d'interprétation des erreurs délivrées par 4D Compiler.
- l'aide à la correction est fournie par le fichier d'erreurs que vous utilisez comme fichier texte ou à l'intérieur de 4^e Dimension.
- l'aide à l'exécution ou contrôle d'exécution vous fournit un outil supplémentaire de contrôle de cohérence et de fiabilité de vos applications.
- l'aide au typage est fournie par le fichier de typage.

La table des symboles

La table des symboles est un document de type texte, plus ou moins long suivant l'importance de vos bases. Il se présente ainsi lorsque vous l'ouvrez à l'aide d'un éditeur de texte :

```
Demo          25/03/97    14:07

Taille des variables interprocess : 12

BFIN  Réel      (F) [Ventes].Affiche_Tab
BOK   Réel      (F) [Ventes].fApropos
BSIMULATION Réel (F) [Ventes].Affiche_Tab
DEJAFAIT Booléen (M) Init_Tableaux
SOMMET1 Entier long (M) Init_Tableaux
SOMMET2 Entier long (M) Init_Tableaux
SOMMET3 Entier long (M) Init_Tableaux
SOMMET4 Entier long (M) Init_Tableaux
TPAYS Chaîne fixe:15 1 dimension (M) Init_Tableaux
TSOMME Entier long 1 dimension (M) Init_Tableaux
TTRIM1 Entier long 1 dimension (M) Init_Tableaux
TTRIM2 Entier long 1 dimension (M) Init_Tableaux
TTRIM3 Entier long 1 dimension (M) Init_Tableaux
TTRIM4 Entier long 1 dimension (M) Init_Tableaux
```

L'en-tête présente le nom de la base, la date et l'heure de la création du document. Le document se compose de quatre parties :

- la liste des variables interprocess.
- la liste des variables process.
- la liste des variables locales dans leur méthode.
- la liste complète des méthodes projet et des méthodes base avec leurs paramètres, le cas échéant.

La liste des variables process et interprocess

Ces deux listes sont séparées en quatre colonnes par des tabulations :

- la première contient la liste complète des variables process et interprocess utilisées dans votre base. Ces variables sont rangées par ordre alphabétique.
- la deuxième affiche le type de la variable. Ce type a été déterminé par une directive de compilation sinon il est déduit par le compilateur en fonction de l'utilisation de la variable. Si le type d'une variable n'a pu être déterminé, cette colonne reste vide.
- la troisième colonne contient le nombre de dimensions du tableau, lorsque la variable est un tableau.
- la quatrième colonne comporte la référence au contexte dans lequel le compilateur a trouvé le type de la variable. Si la variable est utilisée dans plusieurs contextes, seul le contexte qui a permis au compilateur de déterminer son type est mentionné.

Si la variable est trouvée dans une méthode base, son nom est inscrit comme il a été défini dans 4^e Dimension, précédé de (M)*.

Si la variable est trouvée dans une méthode projet, son nom est inscrit comme il a été défini dans 4^e Dimension, précédé de (M).

Si la variable est trouvée dans une méthode table (trigger), c'est le nom de la table qui est inscrit, précédé de (MT).

Si la variable est trouvée dans une méthode formulaire, le nom du formulaire est inscrit, précédé du nom de la table et de (MF).

Si la variable est trouvée dans une méthode objet, le nom de la méthode objet est inscrit précédé du nom du formulaire, du nom de la table et de la mention (MO).

Si la variable est un objet d'un formulaire, sans intervenir dans une quelconque méthode projet, méthode formulaire, méthode table ou méthode objet, le nom du formulaire dans lequel elle apparaît est inscrit, précédé de la mention (F).

En fin de liste, vous trouvez la taille des variables process et interprocess en octets.

Note Sur Macintosh les limites de 32 ko de variables process et 32 ko de variables interprocess sont supprimées. Dans le cas des Power Macintosh et des PC, cette limite n'existe pas. Attention : Dans le cas des variables process, au moment de la compilation, 4D Compiler ne peut déterminer dans quel process une variable de ce type est utilisée. En effet, une variable process peut avoir une valeur différente dans chaque process. Toutes les variables process sont donc systématiquement dupliquées à chaque création de process : la limite des 32 ko n'existant plus, il convient de prendre garde à la taille mémoire qu'elles vont occuper. La taille des variables process est totalement indépendante de celle de la pile des process.

La liste des variables locales

La liste des variables locales apparaît classée par méthode base, méthode projet, méthode table, méthode formulaire et méthode objet en suivant le même ordre que dans 4^e Dimension.

Cette liste est séparée en trois colonnes par des tabulations :

- la première contient la liste des variables locales utilisées dans la méthode ;
- la deuxième affiche le type de la variable ;
- la troisième contient le nombre de dimensions du tableau, lorsque la variable est un tableau.

La liste complète des méthodes

A la fin du fichier sont réunies toutes vos méthodes base et projet avec éventuellement le type de leurs paramètres et du résultat renvoyé.

Ces informations apparaissent sous la forme suivante :
Nom de la méthode(types des paramètres):type du résultat

Le fichier d'erreurs

Un fichier d'erreurs comporte trois types de messages :

- les erreurs générales,
- les erreurs attachées à une ligne précise,
- les warnings.

Ce fichier peut être utilisé de deux façons :

- soit en tant que fichier Texte.
- soit exploité par 4^e Dimension.

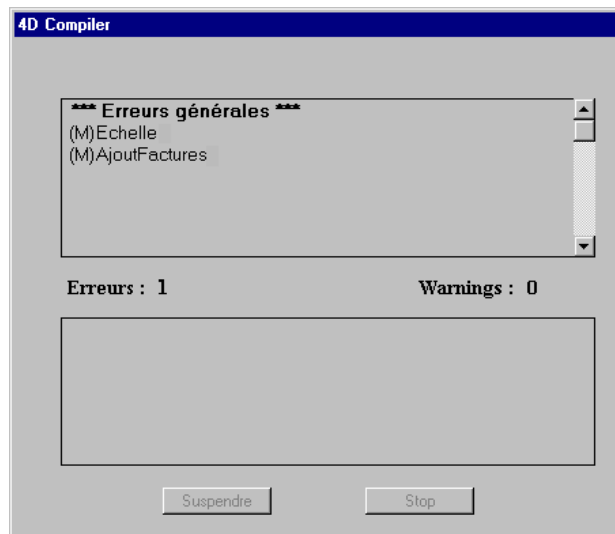
Les types de messages

Les messages que vous délivre 4D Compiler peuvent être classées en trois catégories.

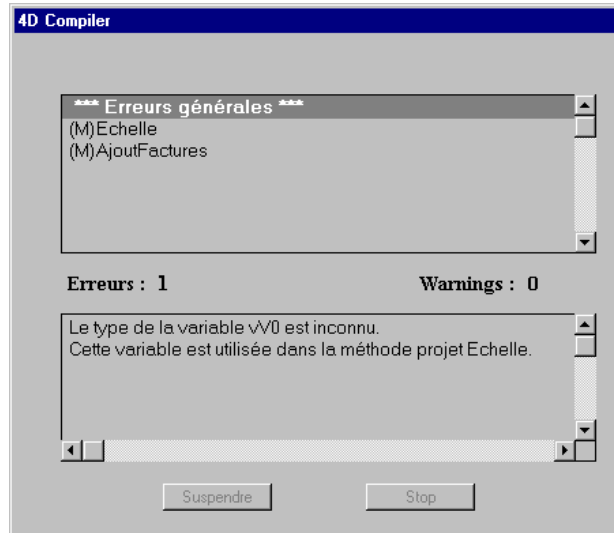
Les erreurs générales

Il s'agit d'erreurs qui ne permettent pas le passage en passe de compilation.

Elles apparaissent en tête de fichier d'erreurs, et, dans le compilateur, elles apparaissent au-dessus de la liste des méthodes.



Si vous cliquez sur **Erreurs générales**, les messages correspondants vous sont délivrés dans la fenêtre inférieure.



Le compilateur délivre une erreur générale dans deux cas :

- si le type d'une variable process ou interprocess n'a pas pu être déterminé.
- si deux objets de nature différente portent le même nom.

Ces erreurs sont dites générales parce qu'elles ne peuvent être rattachées à aucune méthode en particulier.

En effet, 4D Compiler n'a pu procéder au typage nulle part dans la base pour le premier cas, et, dans le second, 4D Compiler ne peut choisir d'associer un nom à un objet plutôt qu'à un autre.

La liste des erreurs générales est fournie dans le [chapitre "Messages", page 123](#).

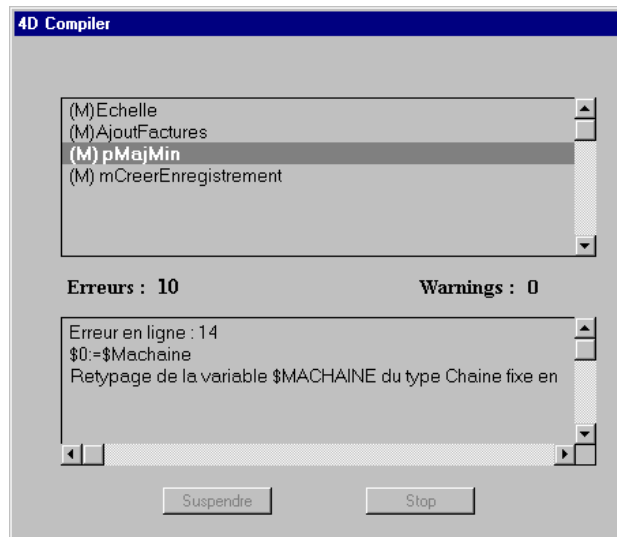
Les erreurs attachées à une ligne précise

Ces erreurs, de différentes natures, vous sont indiquées accompagnées de leur contexte, de la ligne où elles ont été détectées et d'un commentaire.

Elles apparaissent lorsque le compilateur rencontre une expression qui lui pose un problème, que ce soit de typage ou de syntaxe.

Les méthodes base et projet, méthodes table, méthodes formulaire ou méthodes objet où une telle erreur a été détectée apparaissent en gras dans la fenêtre supérieure. Lorsque vous sélectionnez l'une d'elles, la nature de l'erreur vous est détaillée dans la fenêtre inférieure de la façon suivante :

- numéro de la ligne,
- ligne de l'erreur telle qu'elle apparaît dans votre méthode,
- diagnostic sur l'erreur.



La liste de ces erreurs est fournie dans le [chapitre "Messages", page 123](#).

Les warnings

Les warnings ne sont pas des erreurs. Ils n'empêchent en rien la création de la base compilée.

Il s'agit simplement de points sur lesquels le compilateur souhaite attirer votre attention parce qu'il y a un risque d'erreur.

Les méthodes base et projet, méthodes table, méthodes formulaire ou méthodes objet où un warning a été détecté apparaissent en italique dans la fenêtre supérieure.

Lorsque vous sélectionnez l'une d'elle, la nature du warning vous est détaillée dans la fenêtre inférieure de la façon suivante :

- numéro de la ligne,
- ligne du warning telle qu'elle apparaît dans votre méthode,
- raison du warning.

Si vous avez choisi l'option Warning Plus, les messages vous seront délivrés de la même manière.

La liste complète des warnings et des warnings plus vous est donnée dans le [chapitre "Messages", page 123](#).

L'exploitation du fichier d'erreurs

Ainsi que nous l'avons vu plus haut, ce fichier peut être exploité de deux façons : sous la forme d'un document Texte ou à l'intérieur de 4^e Dimension.

Le document exploité sous forme de Texte

Ce document est plus ou moins long suivant le nombre d'erreurs et de warnings délivrés par 4D Compiler. Il se présente ainsi lorsque vous l'ouvrez avec un éditeur de texte :

```
Erreurs : 8      Warnings : 0

*** Erreurs générales ***

Le type de la variable UVER4D est inconnu.
Cette variable est utilisée dans la méthode finformation.
Le type de la variable MOTSCLEFS est inconnu.
Cette variable est utilisée dans la méthode fmotsmoins.
Deux méthodes portent le nom : PROC10
Une variable et une fonction portent le même nom : CALC

Appel

Erreur en ligne : 1
Fct :=4
Erreur de syntaxe
Erreur en ligne : 9
N{1}:=50
La variable N n'est pas un tableau

finformation

Warning en ligne : 5
TABLEAU VERS SELECTION<Ptr>;{Livres}Nom>
Utilisation de pointeur(s) comme paramètre(s) de TABLEAU VERS SELECTION

MiseAJour

Erreur en ligne : 19
$Livraison:=$1
Retypage de la variable $1 du type Réel en type Texte
```

Note Ne soyez pas découragé si votre première compilation génère un nombre impressionnant d'erreurs. En effet, l'interpréteur de 4^e Dimension est très tolérant. Le compilateur réclame de votre part une attention à laquelle vous vous habituerez très vite.

Ce fichier d'erreurs est structuré ainsi :

- en tête de ce fichier, sont indiqués le nombre d'erreurs et le nombre de warnings détectés ;
- sous la rubrique Erreurs générales, sont regroupées toutes les impossibilités de typage et les ambiguïtés d'identité ;
- enfin, regroupées par méthode et cela dans le même ordre que celui que vous avez observé dans 4^e Dimension, 4D Compiler indique toutes les autres erreurs et warnings.

Ces erreurs et warnings sont détaillés de la façon suivante :

- en premier lieu, le numéro de ligne dans la méthode ;
- en second lieu, la ligne concernée par l'erreur ou le warning ;
- en troisième lieu, un diagnostic sur la nature de l'erreur.

Si votre base ne présente aucune erreur générale, le fichier ne comporte pas de section "Erreurs générales". Il mentionne le nombre d'erreurs et de warnings et passe directement aux erreurs relevées méthode par méthode.

```
Erreurs : 8      Warnings : 0

(P) ProcCur1
Erreur en ligne : 1
TABLE PAR DEFAUT
Cette routine nécessite au moins un paramètre.

(S) Table.Page.CalcAge
Erreur en ligne : 2
$MonAnnée:=Annee de($MonAnnée)
Le résultat de la fonction est incompatible avec l'expression.
Erreur en ligne : 2
$MonAnnée:=Annee de($MonAnnée)
Affectation impossible entre ces deux types.
Erreur en ligne : 5
Si <<(<vAge>0)
Cette expression contient trop de parenthèses ouvrantes.
Erreur en ligne : 5
Si <<(<vAge>0)
Erreur de syntaxe
Erreur en ligne : 7
Fin de si
Erreur de syntaxe

(S) Table1.Page.SéITexte
Erreur en ligne : 3
Globale:=Sous_chaine([Table1]Commentaire;$DebutSel;$FinSel-$DebutSel))
Cette expression contient trop de parenthèses fermantes.
Erreur en ligne : 5
Fin de boucle
Erreur de syntaxe
```

L'exploitation sous 4^e Dimension

4D Compiler et 4^e Dimension vous offrent conjointement la possibilité de corriger directement vos erreurs en temps réel.

Générez votre fichier d'erreurs. Pour l'ouvrir dans 4^e Dimension, vous devez :

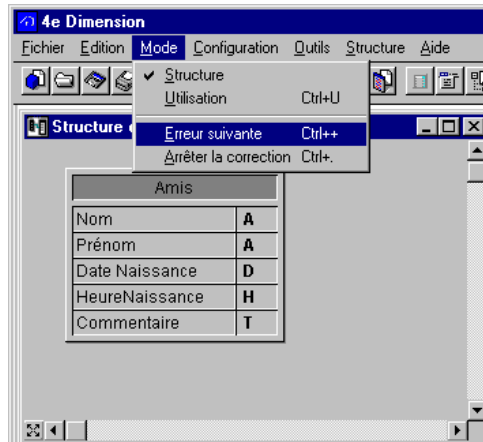
- placer le fichier d'erreurs dans le même dossier que la base ;

Note Sous Windows, 4^e Dimension a également besoin que le fichier nommé NomDeLaBase.ERC soit placé au niveau de la structure de la base pour pouvoir lire le fichier d'erreurs.

- donner à ce fichier le nom que 4^e Dimension propose par défaut, c'est-à-dire NomDeLaBase.err.

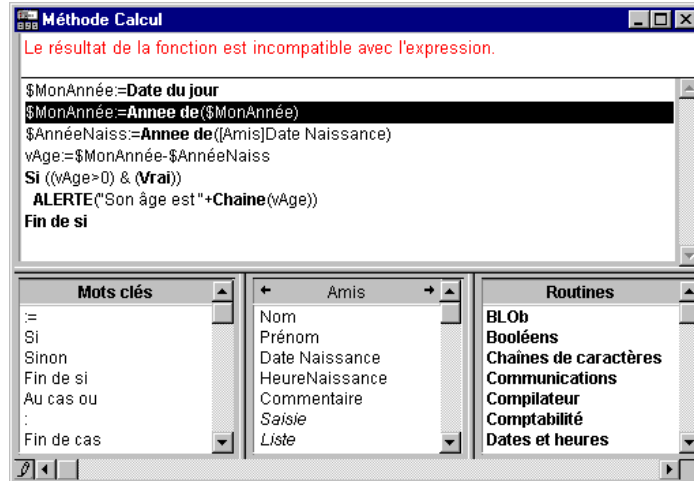
Vous démarrez alors 4^e Dimension et ouvrez votre base non compilée en mode **Structure**.

Deux commandes sont automatiquement ajoutées dans le menu **Mode**, correspondant aux manipulations du mode **Correction d'erreurs**.



Ces deux commandes sont **Erreur suivante** et **Arrêter la correction**. Dès que vous choisissez la commande **Erreur suivante**, 4^e Dimension ouvre la première méthode où 4D Compiler a détecté une erreur ou un warning. Vous accédez alors à l'éditeur de méthodes. Le haut de la fenêtre mentionne la nature de l'erreur ou du warning.

Dans la méthode proprement dite, la ligne concernée apparaît en vidéo inverse.



Lorsque vous avez corrigé votre erreur, vous pouvez passer à l'erreur suivante, que cette erreur soit dans la même méthode ou ailleurs, en demandant **Erreur suivante** dans le menu **Mode**. Vous accédez immédiatement à l'erreur suivante.

Passez ainsi en revue tous les endroits de votre base dans lesquels 4D Compiler a un message à vous délivrer.

Note Il est bien évident que les erreurs générales n'étant rattachées à aucun point précis de la base, elles ne peuvent être affichées de cette manière.

La commande **Arrêter la correction** vous permet à tout moment d'arrêter vos corrections.

Quitter le mode Structure ferme automatiquement le fichier des erreurs.

Le fichier de typage

Ce document de type texte se présente comme suit :

```
C_REEL<@A>

C_TEXTE<B>
C_BOOLEAN<C>

` (M) Réponse1
C_TEXTE<REPONSE1;$1>
C_TEXTE<REPONSE1;$2>

` (M) Réponse2
C_REEL<REPONSE2;$0>
C_REEL<REPONSE2;$1>

` (M) PROC1
C_ENTIER LONG<PROC1;$1>

` (M) PROC2
C_ENTIER LONG<PROC2;$1>

(M) Réponse1
C_TEXTE<$1>
C_TEXTE<$2>

(M) Réponse2
C_REEL<$0>
C_REEL<$1>

(M) PROC1
C_ENTIER LONG<$1>

(M) PROC2
C_ENTIER LONG<$1>
```

Le document se compose de 4 parties :

- les directives de compilation pour les variables interprocess,
- les directives de compilation pour les variables process,
- les directives de compilation pour les variables locales,
- les directives de compilation pour les paramètres.

Ce fichier sera une aide précieuse si vous souhaitez réduire le chemin de compilation. Il vous suffira alors de coller son contenu dans les méthodes de typage :

- pour les variables interprocess, process et les paramètres, dans les méthodes de typage dont le nom commence par COMPILER ;
- pour les variables locales, à l'intérieur des méthodes où elles sont utilisées.

Pour plus de précisions à ce sujet, reportez-vous au paragraphe [“Où placer vos directives de compilation ?”, page 33.](#)

Le contrôle d'exécution

Définition et fonction du contrôle d'exécution

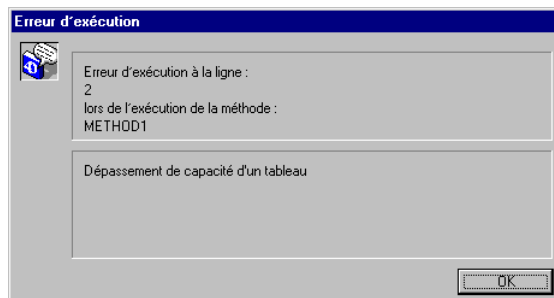
Alors que toutes les autres options fonctionnent durant le processus de compilation, le contrôle d'exécution fonctionne lorsque vous lancez une base compilée. C'est donc en cours d'exécution de votre application que vous recevrez les messages délivrés par 4D Compiler.

Le contrôle d'exécution introduit une analyse supplémentaire par rapport à la recherche de cohérence logique et syntaxique qui caractérise normalement un compilateur. Dans le contrôle d'exécution, le compilateur introduit pour vous la question suivante : "compte tenu de ce que vous m'avez demandé, j'arriverai probablement à un résultat qui peut vous surprendre". Le contrôle d'exécution est un contrôle en situation, dépendant de l'état des objets de la base à un instant donné.

Imaginez, par exemple, que dans votre base, vous déclariez un tableau LeTableau de type Texte. Suivant la méthode en cours d'exécution, le nombre d'éléments de LeTableau peut varier. Vous souhaitez affecter à l'élément 5 du tableau la valeur "Bonjour". Vous écrivez donc : `LeTableau{5}:="Bonjour"`.

Si, à ce moment-là, LeTableau comporte 5 éléments ou plus, tout va bien. L'affectation s'effectue normalement. Il y a bien un élément numéro 5, prêt à recevoir cette valeur. Si, pour une raison ou pour une autre, à ce moment précis de l'exécution, LeTableau comporte moins de 5 éléments, 2 par exemple, votre affectation n'a plus de sens.

Il est évident que ce type de situation ne peut être détecté à la compilation puisqu'il présuppose l'exécution des méthodes. Seul le contrôle d'exécution vous permet d'avoir la maîtrise de ce qui se passe effectivement lorsqu'on utilise la base. Dans l'exemple évoqué ci-dessus, le compilateur vous délivrera donc, dans 4^e Dimension, un message du type :



On comprend dès lors que le contrôle d'exécution soit particulièrement précieux dans le cas de manipulation sur les tableaux, les pointeurs et les chaînes de caractères.

Les messages type délivrés par 4D Compiler lorsque vous demandez le contrôle d'exécution sont indiqués dans le [chapitre "Messages", page 123](#).

Note A l'attention des programmeurs en Pascal, notre contrôle d'exécution correspond à un Range Checking dont les fonctionnalités seraient étendues.

Quand et comment tirer parti du contrôle d'exécution ?

Lorsque vous demandez le contrôle d'exécution, votre base compilée est sensiblement moins rapide et plus volumineuse que lorsque vous n'avez pas sélectionné cette option. Ne vous privez pas pour autant de cet outil puissant qui consiste finalement à simplement introduire une étape supplémentaire dans le processus de développement et de debugging de vos bases, puisqu'il est entendu que lorsque tout le travail sera terminé et votre base parfaitement testée, vous ferez une compilation finale sans contrôle d'exécution.

La finalité la plus évidente du compilateur est d'accélérer les bases. Le but, non moins important, que s'est fixé 4D Compiler est, de surcroît, de vous aider à fiabiliser vos applications. Nous vous recommandons donc d'avoir toujours une phase avec le contrôle d'exécution.

Pour mémoire : en cas d'anomalies

Ce paragraphe est une justification *a contrario* de l'utilité du contrôle d'exécution. Vous notez des anomalies de fonctionnement dans vos bases ? Avant toute spéculation sur leur origine, pensez aux aides fournies par le compilateur.

Les anomalies que vous pourriez avoir sont les suivantes :

- 4^e Dimension fait apparaître ses propres messages d'erreurs et vous n'en voulez pas.
Corrigez cette erreur dans votre application d'après l'indication fournie par 4^e Dimension. Si cette indication vous semble trop générale, recompilez votre base en sélectionnant le contrôle d'exécution. Testez à nouveau votre base : à l'endroit où apparaissait le message de 4^e Dimension, vous aurez un message délivré par le compilateur.

- Votre base compilée ne fonctionne pas exactement comme votre base non compilée. Examinez d'un peu plus près les messages des Warnings et Warnings plus. Eventuellement, ajoutez l'option de contrôle d'exécution.
- Votre base fonctionne normalement en mode interprété et, lorsque vous l'utilisez après la compilation, une erreur provoque un "plantage" du système. Une solution simple et agréable est de compiler avec l'option "Contrôle d'exécution". Il y a toutes les chances que votre problème soit détecté et donc en bonne voie de résolution.
- Votre base fonctionne en mode interprété et une erreur système se produit en mode compilé. Vérifiez que votre base compilée dispose des mêmes plug-ins que ceux que vous avez utilisés au moment de la compilation.
- Les variables Numériques ou Alphanumériques ne présentent pas les valeurs que vous attendiez. Vérifiez les options de typage par défaut de votre projet de compilation et générez la table des symboles pour vérifier que toutes vos variables sont bien typées comme vous le souhaitez.

8

Messages

Dans cette section, nous allons détailler les messages qui vont permettre à 4D Compiler de dialoguer avec vous. Ainsi que nous l'avons déjà vu, les messages que le compilateur vous délivre durant la compilation sont de plusieurs types :

- les **Warnings**, délivrés durant la passe de typage directif ;
- les **Warnings Plus**, qui vous aident à déjouer des pièges facilement évitables ;
- les **erreurs**, qu'il vous appartient de corriger ;
- les **messages de contrôle d'exécution**, délivrés dans 4^e Dimension ;
- les **messages d'alerte**, messages de contrôle de l'environnement.

Note La liste de ces messages n'est peut-être pas complète puisqu'elle a été établie au moment des tests de 4D Compiler.

Les warnings

Ces messages vous sont délivrés lors du typage des variables de la base à compiler. Chaque message est accompagné ici d'un exemple de ce qui a pu le provoquer et, éventuellement, d'une explication supplémentaire.

Utilisation de pointeur(s) comme paramètre(s) de COPIER TABLEAU
COPIER TABLEAU(LePointeur->;LeTableau)

Utilisation de pointeur(s) comme paramètre(s) de SELECTION VERS TABLEAU
SELECTION VERS TABLEAU(LePointeur->;LeTableau)
SELECTION VERS TABLEAU([LaTable]LeChamp;LePointeur->)

Utilisation de pointeur(s) comme paramètre(s) de TABLEAU VERS SELECTION
TABLEAU VERS SELECTION(LePointeur->[LaTable]LeChamp)

Utilisation de pointeur(s) comme paramètre(s) de ENUMERATION VERS TABLEAU
ENUMERATION VERS TABLEAU(Enum;LePointeur->)

Utilisation de pointeur(s) comme paramètre(s) de TABLEAU VERS ENUMERATION
TABLEAU VERS ENUMERATION(LePointeur->;Enum)

Utilisation de pointeur(s) dans une déclaration de tableau
TABLEAU REEL(LePointeur->;5)

L'instruction **TABLEAU REEL**(LeTableau;LePointeur->) ne provoquera pas cet avertissement. La valeur de la dimension d'un tableau n'a pas d'influence sur son type.

Utilisation de pointeur(s) comme paramètre(s) de VALEURS DISTINCTES
VALEURS DISTINCTES(LePointeur->;LeTableau)

Utilisation de la fonction Indefinie
Si(Indefinie(LaVariable))

Cette méthode est protégée par un mot de passe.

Le Formulaire LeFormulaire contient un bouton avec action sans nom dans la page 1.

Tous vos boutons avec action doivent avoir un nom afin d'éviter des conflits.

Les warning plus

Ces messages vous sont délivrés tout au long du processus de compilation. Chaque message est accompagné ici d'un exemple de ce qui a pu le provoquer et éventuellement, d'une explication supplémentaire.

Le pointeur utilisé dans cette commande doit pointer sur un Alphanumérique.

LePointeur->≤2≥:="a"

Le pointeur utilisé dans cette commande doit pointer sur un Entier, un Entier long ou un Numérique

LaChaine≤LePointeur->≥:="a"

Un indice de tableau doit être Entier, Entier long ou Numérique.
ALERTE(LeTableau{LePointeur->})

Il manque un paramètre à l'appel de la commande du plug-in.
sp_FIXER FORMAT(LaZone)

Les erreurs

Ces messages vous sont délivrés tout au long du processus de compilation.

Chaque message est accompagné ici d'un exemple de ce qui a pu le provoquer et éventuellement, d'une explication supplémentaire.

Il vous appartient de corriger ces erreurs afin de permettre à 4D Compiler de générer une base compilée.

Ces messages sont classés thématiquement. Les thèmes dans lesquels vous pourrez les consulter sont les suivants :

- [Typage](#)
- [Syntaxe](#)
- [Paramètres](#)
- [Opérateurs](#)
- [Plug-ins](#)
- [Erreurs générales](#)

Typage

Cet opérateur ne peut s'appliquer à ce type de variable.

Cette affectation provoquerait un conflit de type.

LeRéal:=12,3

LeBooléen:=**Vrai**

LeRéal:=LeBooléen

Changement de la longueur maximale d'une chaîne de caractères.

C_ALPHA(3;LaChaîne)

C_ALPHA(5;LaChaîne)

Changement du nombre de dimensions d'un tableau.

TABLEAU TEXTE(LeTableau;5;5)

TABLEAU TEXTE(LeTableau;5)

Conflit de type sur la variable LeTableau dans le formulaire LeFormulaire.
TABLEAU ENTIER(LeTableau)

Déclaration d'un tableau sans indice.
TABLEAU ENTIER(LeTableau)

Il manque une variable.
COPIER TABLEAU(LeTableau;"")

Il manque une constante.
C_ALPHA(LaVariable;LaChaine)

*Impossible de déterminer le type de LaVariable.
Cette variable est utilisée dans la méthode M1.*
Le type de LaVariable n'a pu être déterminé. Une directive de compilation sera nécessaire.

Je n'attendais pas une constante de type Texte.
OK := "Il fait beau"

La méthode M1 est inconnue.
La ligne contient un appel à une méthode qui n'existe pas ou plus.

Le champ utilisé dans cette expression provoque un conflit de type.
MaDate:=**Ajouter a date**(LeChampBool;1;1;1)

La taille d'une chaîne de caractères doit être comprise entre 1 et 255.
C_ALPHA(325;LaChaine)

La variable LaVariable n'est pas une méthode.
LaVariable(1)

La variable LaVariable n'est pas un tableau.
LaVariable{5}:=12

Le résultat de la fonction est incompatible avec l'expression.
LeTexte:="Numéro"+**Num**(i)

Les types utilisés dans l'expression sont incompatibles.
LEntier:=LaDate*LeTexte

Utilisation de la variable \$i de type alphanumérique comme variable de type réel.
\$i:="3"
\$(\$i):=5

L'indice du tableau n'est pas numérique.
TabEntier{"3"}:=4

Retypage de la variable LaVariable du type Texte en tableau de type Texte.

C_TEXTE(LaVariable)

COPIER TABLEAU(TabTexte;LaVariable)

Retypage de la variable LaVariable du type Texte en type Réel.

LaVariable:=**Num**(LaVariable)

Retypage du tableau LeBooléen du type Booléen en variable de type Réel.

LaVariable:=LeBooléen

Retypage du tableau du type Entier en type tableau de type Texte.

TABLEAU TEXTE(TabEntier;12)

si TabEntier a été déclaré ailleurs comme tableau d'Entiers.

Seuls les pointeurs peuvent être suivis du signe ->

LaVariable->:=5

si la variable n'est pas de type Pointeur.

Utilisation de la variable LaVar1 de type Texte comme une variable de type Numérique.

LaVar1:=3,5

Utilisation d'un champ de type incorrect.

[LaTable]LeChamp est un champ de type Date.

LaVariable est de type Numérique.

LaVariable:=[LaTable]LeChamp

Syntaxe

Cette fonction ne retourne pas un pointeur.

LaVariable:=**Num**("Il fait beau")->

Il n'est pas possible de dépointer cette fonction.

Erreur de syntaxe.

Si(LeBooléen)

Fin de boucle

Il manque une accolade fermante.

La ligne comporte plus d'accolades ouvrantes que d'accolades fermantes.

Il manque une accolade ouvrante.

La ligne comporte plus d'accolades fermantes que d'accolades ouvrantes.

Il manque une parenthèse fermante.

La ligne comporte plus de parenthèses ouvrantes que de fermantes.

Il manque une parenthèse ouvrante.

La ligne comporte plus de parenthèses fermantes que d'ouvrantes.

J'attendais un champ.

Si(Modifie(LaVariable))

J'attendais une accolade ouvrante.

C_ENTIER(\$

J'attendais une variable.

C_ENTIER([LaTable]LeChamp)

J'attendais un nombre constant.

C_ENTIER({ "3" })

J'attendais un point virgule.

COPIER TABLEAU(LeTableau1 LeTableau2)

■ Sous MacOS

J'attendais le signe ≥

LaChaine≤3:="a"

J'attendais le signe ≤

LaChaine3≥:="a"

■ Sous Windows

J'attendais le signe]]

LaChaine[[3:="a"

J'attendais le signe [[

LaChaine 3]]:="a"

Je n'attendais pas un champ de type sous-table

TABLEAU VERS SELECTION(LeTableau;Sous-table)

Le type du paramètre de Si doit être booléen.

Si(LePointeur)

L'expression est trop complexe.

Divisez votre ligne en plusieurs sous-opérations.

Méthode trop complexe.

Au cas ou avec plus de 600 cas différents ou plus de 100 Si...Fin de si imbriqués.

Référence à un champ inconnu.

Votre méthode, probablement copiée d'une autre base, contient •???• à la place d'un nom de champ.

Référence à une table inconnue.

Votre méthode, probablement copiée d'une autre base, contient •???• à la place d'un nom de table.

Un pointeur ne peut être défini sur cette expression.

LePointeur:=->LaVariable+3

Utilisation incorrecte des indices de chaînes de caractères.

LeNumérique≤3 ou LeNumérique [[3]]

ou bien

LaChaine≤LaVariable ou LaChaine[[LaVariable]]

où LaVariable n'est pas une variable Numérique.

Paramètres

Ce résultat de fonction ne peut pas être paramètre de cette méthode.

LaMéthode(Num(LaChaine))

si LaMéthode attend un paramètre de type Booléen.

Cette routine reçoit trop de paramètres

TABLE PAR DEFAULT(LaTable;LeFormulaire)

Cette valeur ne peut pas être paramètre de cette méthode.

LaMéthode(3+2)

si LaMéthode attend un paramètre de type Booléen.

Conflit de type sur la variable \$0.

C_ENTIER(\$0)

\$0:=Faux

Conflit de type sur le paramètre générique.

C_ENTIER(\$ {3})

Boucle(\$i;3;5)

\${i}:=Chaine(\$i)

Fin de boucle

La routine n'attend pas de paramètre.

AFFICHER BARRE OUTILS(MaVar)

La routine nécessite au moins un paramètre.

TABLE PAR DEFAULT

La variable LaChaine ne peut pas être paramètre de cette méthode.

LaMéthode(LaChaine)

si LaMéthode attend un paramètre de type Booléen.

Le type du paramètre \$1 dans la méthode est différent de celui du paramètre à l'appel.

Calcul("3+2")

avec la directive C_ENTIER(\$1) dans Calcul.

Le type du paramètre passé ne correspond pas au type du paramètre attendu.

Impri("LaserWriter")

si dans la méthode Impri, \$1 est de type Numérique.

L'un des paramètres de COPIER TABLEAU est une variable.

COPIER TABLEAU(LaVariable;LeTableau)

Retypage du paramètre \$1 du type réel en type Texte.

\$1:=Chaine(\$1)

Un paramètre ne peut être un tableau.

Rélnit(LeTableau)

Pour passer un tableau à une méthode, il faut passer un pointeur sur ce tableau.

Un paramètre ne peut être utilisé dans l'appel de cette routine.

RECEVOIR VARIABLE(\$1)

Utilisation du paramètre \$1 de type Booléen comme une variable de type Entier.

PROPRIETES CHAMP(NoDeTable;NoDeChamp;Type;\$1)

Opérateurs

Cet opérateur ne peut s'appliquer à ce type de variable.

LeBool2:=LeBool1+**Vrai**

L'addition ne peut pas s'appliquer à des Booléens.

Je n'attendais pas l'opérateur >

CHERCHER(LaTable;[LaTable]LeChamp=0;>)

Les deux opérandes ne sont pas comparables.

Si(LeLongE=Image2)

Le signe moins ne peut pas être utilisé dans cette expression.

LeBool:=-**Faux**

Plug-ins

La commande PExt du plug-in est mal définie.

Il manque des paramètres à l'appel de la commande du plug-in.

Le nombre de paramètres envoyés à la commande du plug-in est trop grand.

La commande LaVariable du plug-in est mal définie.

Erreurs générales

Deux méthodes portent le même nom : LeNom.

Pour pouvoir compiler votre base, il faut que toutes les méthodes projets aient des noms différents.

Erreur interne n° xx.

Au cas où ce message apparaîtrait dans l'une de vos bases, téléphonez au support technique de 4D et signalez le numéro de l'erreur.

Je n'ai pas pu déterminer le type de LaVariable. Cette variable est utilisée dans la méthode M1.

Le type de LaVariable n'a pu être déterminé. Une directive de compilation est nécessaire.

La base contient plus de 32 Ko de variables locales : 40 Ko.

(pour tous les types de machines)

Il vous faut réduire le nombre de variables locales de la méthode.

Le compilateur vous indique la quantité de mémoire occupée par vos variables locales dans cette méthode, hors chaînes fixes (déclarées par la directive C_ALPHA).

La méthode dépasse 32 Ko.

(Toutes les machines équipées de processeur 680XX)

Votre méthode dépasse la limite des 32 Ko autorisés. Scindez-là en plusieurs sous-méthodes.

La méthode originale est endommagée.

La méthode est endommagée dans la structure originale. Supprimez-la ou remplacez-la.

Méthode 4^e Dimension inconnue.

La méthode est endommagée.

Retypage de la variable LaVariable dans le Formulaire LeFormulaire.

Ce message apparaîtra si vous donnez, par exemple, le nom OK à une variable de type Graphe dans un formulaire.

Une fonction et une variable portent le même nom : LeNom.

Renommez soit la fonction, soit la variable.

Une variable dessinée dans le Formulaire LeFormulaire a le même nom qu'une fonction : LeNom.

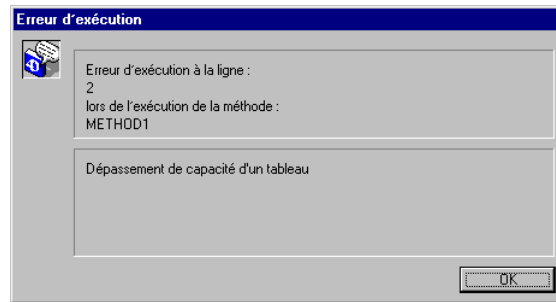
Renommez soit la fonction, soit la variable.

Une méthode et une variable portent le même nom : LeNom.
Renommez soit la méthode, soit la variable.

Une commande du plug-in et une variable portent le même nom : LeNom.
Renommez soit la commande du plug-in, soit la variable.

Les messages du contrôle d'exécution

Ces messages vous sont délivrés dans 4^e Dimension, lors de l'utilisation de la base compilée. Ils sont affichés dans 4^e Dimension dans la fenêtre suivante :



Dépassement de capacité d'un tableau.

LeTableau est un tableau à 5 éléments à un instant donné. Ce message apparaîtra si vous essayez d'accéder à l'élément LeTableau{17} à cet instant.

Division par zéro.

Var1:=0

Var2:=2

Var3:=Var2 / Var1

Le paramètre utilisé n'a pas été passé.

Utilisation de la variable \$4 alors que seuls trois paramètres ont été passés lors de l'appel courant.

Le pointeur n'est pas correctement initialisé.

LePointeur->:=5

si LePointeur n'a pas encore été initialisé.

La chaîne dans laquelle se fait l'affectation est trop courte.

C_ALPHA(LaChaine1;5)

C_ALPHA(LaChaine2;10)

LaChaine2 :="Bonjour"

LaChaine1:= LaChaine2

L'indice de chaîne n'est pas valide (trop grand ou négatif).

i:=-30

LaChaine≤i≥:= LaChaine2 ou LaChaine[[i]]:=LaChaine2

La chaîne passée en paramètre est vide ou non initialisée.

LaChaine≤1≥:= ""

LaChaine[[1]]:= ""

Modulo par zéro.

Var1:=0

Var2:=2

Var3:=Var2 % Var1

Paramètres incorrects dans une commande EXECUTER.

EXECUTER("MaMéthode(MonAlpha)")

si MaMéthode attend un paramètre autre qu'alphanumérique.

Pointeur sur une variable inconnue du compilateur.

LePointeur:= **Pointeur vers** ("LaVariable")

LePointeur:= "MaChaîne"

si LaVariable n'apparaît pas explicitement dans la base.

Tentative de retypage par l'intermédiaire d'un pointeur.

LeBooleen:=LePointeur->

si LePointeur référence un champ de type Entier.

Utilisation incorrecte d'un pointeur.

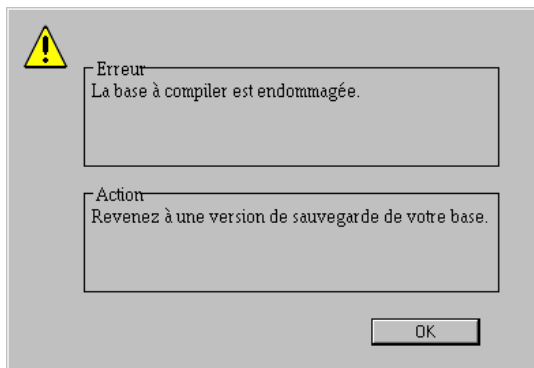
LeCaractère:=LaChaine ≤LePointeur->≥ ou

LeCaractère:=LaChaine[[LePointeur]]

si LePointeur ne référence pas un Numérique.

Les messages d'alerte

Lorsque son environnement de travail n'est pas optimal, 4D Compiler délivre des messages qui vous permettent d'améliorer cet environnement. Ces messages apparaissent dans la fenêtre suivante :



La zone **Erreur** décrit le problème qui est apparu.

La zone **Action** décrit l'opération qui n'a pu être exécutée et éventuellement le moyen d'y remédier.

A

Personnaliser son application

4D Compiler vous donne la possibilité de fusionner la structure de votre base 4^e Dimension avec le fichier 4D Engine souhaité, afin d'obtenir une application exécutable (et double-cliquable), soit pour les plates-formes Macintosh/Power Macintosh, soit pour les plates-formes PC. Cette application peut être personnalisée.

Personnaliser son application sous Mac™ OS

Afin de personnaliser votre application compilée sous MacOS, vous pouvez modifier la signature et l'icône de l'application nouvellement créée. Pour cela, vous devez utiliser un éditeur de ressources de type ResEdit®.

Changer la signature de l'application

Pour changer la signature de votre application, deux opérations sont nécessaires :

- Premièrement, ouvrez la ressource de type BNDL. Remplacez les 4 caractères de la signature de l'application par les 4 caractères que vous aurez choisi pour être la nouvelle signature de votre application. Attention, cette signature devra être unique afin d'éviter tout risque de confusion au lancement de l'application. Si vous repreniez la signature d'un autre programme, le Finder risquerait d'intervertir les différentes icônes.

- Deuxièmement, ouvrez la ressource SIG*. Recopiez dans cette ressource les 4 caractères que vous venez d'attribuer à la ressource BNDL.
- Si votre éditeur de ressources ne l'a pas fait automatiquement, créez une ressource dont le type sera le même que la signature que vous aurez donnée à votre application. Cette ressource, dont le numéro d'identification sera 0, contiendra, par exemple, le nom de votre application.

Changer l'icône de l'application

L'icône de votre application se trouve dans les ressources icl8, icl4, ICN#, ics#, ics4 et ics8.

- 1 Ouvrez l'icône portant le numéro d'identification 128.



- 2 A l'aide de l'éditeur de ressources, modifiez cette icône à votre convenance.

Si vous vous arrêtez là, votre application est prête à être compilée, mais les fichiers créés par la base compilée présenteront les icônes standard de 4^e Dimension. Vous pouvez les voir, par défaut, aux autres numéros d'identification.

Changer les icônes des fichiers annexes

Il est également possible de changer l'icône du fichier .data ainsi que celle des autres fichiers générés par votre base.

- 1 Ouvrez à nouveau la ressource icl4.
Chacune des icônes de la liste correspond à un type de fichiers générés par 4^e Dimension : .data, export ASCII...
- 2 Modifiez ces icônes à votre convenance.
Cette modification ne sera effective que pour les fichiers créés avec la base compilée et personnalisée, les fichiers antérieurs garderont leur icône.

3 Après avoir personnalisé icône et signature de votre application, compilez-la en sélectionnant l'option **Création d'un exécutable**.

Note Si vous souhaitez commercialiser votre application, il vous faudra, au préalable, vous conformer aux modalités fixées par Apple. Vous aurez ainsi l'assurance que votre signature est unique et ne générera pas de conflit d'icône.

Vos modifications ne s'effectueront peut-être pas en temps réel. Dans ce cas, il vous faudra procéder à la reconstruction de votre bureau avant de pouvoir les visualiser. Pour cela, redémarrez votre Macintosh en maintenant les touches **Commande** et **Option** enfoncées.

Personnaliser son application sous Windows

Comme nous l'avons signalé dans le [paragraphe "Création d'un exécutable", page 89](#), il est conseillé de prévoir un installateur pour tout ce qui concerne la personnalisation des icônes de votre application.

Au moment de l'installation du logiciel, l'installateur enregistre les informations qui permettent ensuite au Gestionnaire de programmes d'afficher des icônes.

Toutefois, vous devez auparavant créer une icône. Vous pouvez la créer à l'aide d'un logiciel graphique tel que *Image Editor*, qui est fourni avec le Win32™ SDK de Microsoft® Corporation ou équivalent. Ensuite, vous devez vous procurer un éditeur de ressources Windows pour éditer vos ressources sur PC.

Note Par exemple, vous pouvez vous procurer Resource Workshop de Borland International, Inc.

Changer l'icône de l'application

Pour cette opération, vous devez placer le fichier de votre icône "NomDeLaBase.ico" au même niveau que votre fichier structure "NomDeLaBase.4DB".

1 Choisissez l'option "Création d'un exécutable".

La boîte de dialogue d'ouverture de document apparaît.

2 Sélectionnez le fichier 4D Engine.

3 Cliquez sur OK.

Pendant la compilation, 4D Compiler intègre votre icône au fichier 4D Engine.exe. En fin de compilation, vous obtenez un exécutable nommé "NomDeLaBase.exe" ainsi que les fichiers de votre structure compilée.

Changer le nom de la fenêtre de l'application

Quand vous créez une application exécutable avec 4D Compiler sous Windows, par défaut, le titre de la fenêtre principale correspond au nom de votre fichier. exe.

Par exemple, dans votre répertoire DISTRIB, vous avez : MonNom.4DC, MonNom.4RSR, MonNom.EXE, ASINTPPC.DLL, etc. Le nom de la fenêtre principale sera MonNom.

Si vous voulez un titre de fenêtre personnalisé, vous avez deux choix : une personnalisation sous Mac OS et une sous PC.

Mac OS

- 1 Vous transportez avec 4D Transporter la structure de votre base avant compilation sous Mac OS.
- 2 A l'aide d'un éditeur de ressources, vous ouvrez la structure de votre base.
- 3 Créez une ressource de type "STR " à l'ID 13010.
- 4 Saisissez dans le contenu de la ressource le nom que vous voulez donner au titre de la fenêtre.
- 5 Repassez votre base sur la plate-forme PC.

Au lancement, la fenêtre principale aura pour titre le nom que vous lui avez donné.

Windows

- 1 Ouvrez votre structure avec 4^e Dimension.
- 2 Créez une nouvelle méthode.
- 3 Utilisez la commande ECRIRE RESSOURCE CHAINE pour créer une ressource "STR " d'ID 13010 dans votre structure.

Par exemple :

ECRIRE RESSOURCE CHAINE (13010; "Mon Application")

Après avoir exécuté votre méthode, compilez votre base et créez un exécutable. Au lancement de l'application exécutable, la fenêtre principale aura pour titre "Mon Application".

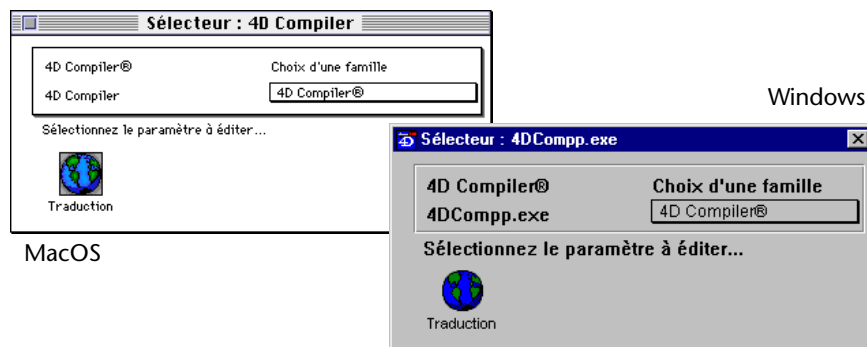
B

Personnaliser 4D Compiler

4D Compiler possède des ressources qui peuvent être personnalisées par Customizer Plus, un utilitaire fourni avec 4^e Dimension.

Note Pour plus d'informations sur le logiciel Customizer Plus, reportez-vous à la documentation de ce programme, fournie au format Acrobat.

Lorsque vous ouvrez l'application 4D Compiler, la fenêtre suivante apparaît :



Elle comporte une seule icône : Traduction.

Traduction

Lorsque vous double-cliquez sur l'icône Traduction, la fenêtre suivante apparaît. Elle présente une liste déroulante :



Cette boîte de dialogue vous permet de traduire la langue des commandes de 4^e Dimension utilisées par 4D Compiler.

Dans la liste déroulante, sélectionnez la langue de votre choix : les fichiers d'erreur et de typage, ainsi que la fenêtre d'erreurs dans le compilateur afficheront les commandes de 4^e Dimension dans cette langue.

Index

Symboles

- .4DC (extension d'une structure compilée) 90
- .EXE (extension d'un exécutable) 90
- .prj (projet par défaut Windows) 83
- .proj (projet par défaut MacOS) 83
- .RSR (extension d'une structure compilée) 90
- .sym (extension Windows table des symboles) 93
- .symb (extension MacOS table des symboles) 93

Chiffres

- 1919382119 (valeur aberrante) 97
- 4D Compiler
 - Lancer le programme 81
 - Personnaliser 139
 - Présentation 9
- 4D Engine 91
 - Fichiers nécessaires 89
- 4D InstallMaker (Installeur) 90
- 4DK#
 - Types 74

A

- Acrobat (Reader) 10
- Aides à la compilation 109
- Alphanumériques
 - Type par défaut 100
- APPELER SUR EVENEMENT 64
- APPELER SUR PORT SERIE 64
- Application compilée
 - Personnaliser 135

B

- Base compilée
 - Utilisation 107
- Boutons
 - Types par défaut 98

C

- Chaînes alphanumériques
 - Optimisation 78
 - Précisions de syntaxe 61
- Champs
 - Optimisation 79
- Changer le nombre de dimensions d'un tableau 43
- Chemin de compilation 97
- Choix du code 95
- Code généré 13
- Commandes de directives de compilation 27
- Commentaires (traitement) 76
- Commercialiser votre application 137
- Communication (précisions de syntaxe) 62
- Compilateur (Définition) 12
- Compilation
 - Aides 109
 - Analyse syntaxique 15
 - avec les routines externes OLE (Windows) 53
 - Chemin 97
 - Commentaires 76
 - copie de la base 103
 - Directives de 27
 - Exécution 101, 105
 - Exemple 18
 - Fenêtres d'options 87
 - Fichiers nécessaires à la fusion 89
 - Interruption 106
 - mode Fat binary 16
 - multi-plate-forme 50
 - Objets compilés 13
 - Principes 23
 - Processus 102
 - Stop 103
 - Suspendre 103
 - Typage des variables 103
 - Vérification du code 15
- Compilé, interprété
 - Différence 12
 - Vitesse d'exécution 15
- Composants 4D 55

Conflits

- entre deux directives de compilation 41
 - entre une utilisation et une directive de compilation 39
 - Nombre de dimensions d'un tableau 43
 - par retypepage implicite 40
 - tableaux de chaînes fixes 44
 - type des éléments d'un tableau 43
 - variables simples 39
 - variables Tableau 42
- Constantes 60, 74
- Contrôle d'exécution 93
- Définition 120
 - Messages 132
 - Utilisation 121
- Copie de la base 103
- COPIER TABLEAU 66
- Création d'un exécutable 53, 89
- Création de la table des symboles 25
- Customizer Plus 139
- Traduction 140

D

- Déduction du type des variables 26
- Directives de compilation
- C_Alpha 35
 - Commandes 27
 - Conflits 41
 - Optimisation du code 76
 - Où les placer 33
 - Quand les utiliser 28
- DISTRIB (répertoire) 89, 90
- Document (variable système) 60
- Documents système (précisions de syntaxe) 63
- Drag and drop 86
- Duplication de la base 103

E

- ECRIRE BLOC 72
- EFFACER VARIABLE 72
- Effectuer les passes de typage 98
- Enregistrer (menu Fichier) 85
- Enregistrer sous... (menu Fichier) 85

Erreurs

- attachées à une ligne précise 113
 - courantes 121
 - Exploitation du fichier 115
 - Fichier 91, 112
 - générales 112
- Erreurs (messages) 125
- Error (variable système) 60
- Etats semi-automatiques
- Variables 60
- Exécutable
- Création 53, 89
- EXECUTER 70
- Exécution (Contrôle) 93
- Exécution d'une compilation 101
- Exemple de compilation 17

F

- Fat binary 16
- Fenêtres des options 81
- de compilation 87
 - secondaire 96
- Fermer (menu Fichier) 85
- Fichier d'erreurs 91, 112
- Arrêter la correction 117
 - Erreur suivante 117
 - Exploitation 115
 - sous 4e Dimension 117
 - sous forme de Texte 115
- Fichier de typage 97, 119
- FldDelimit (variable système) 60
- Fonction Indéfinie 71
- Fonctions mathématiques (précisions de syntaxe) 64
- Formulaires
- Typage des variables dessinées 45
 - Variables considérées comme des Numériques 45
 - Variables considérées comme du Texte 47
 - Variables Graphe 46
 - Variables Zone externe 46
- Fusionner la base avec 4D Engine 91

G

- Graphe (variable) 46

H

- Hypertexte 10

I

Icône de l'application (modifier) 136, 137
 Icône des fichiers annexes (modifier) 136
 Indefinie 71
 Indirections sur les paramètres 58
 Initialisation des locales 96
 Interprété, compilé
 Différence 12
 Vitesse d'exécution 15
 Interruption du processus de compilation 106
 Interruptions (précisions de syntaxe) 64

K

KeyCode (variable système) 60

L

LIRE VARIABLES 72

M

MAC4DX 49
 MacOS
 Compilation multi-plate-forme 51
 MacOS (version) 10
 Manipulation des variables locales \$0...\$N 56
 Manuel 4D Compiler
 Acrobat 10
 Navigation hypertexte 10
 Présentation 10
 Windows, MacOS 10
 Menus de 4D Compiler 81
 Messages 123
 Alertes 134
 Contrôle d'exécution 132
 Erreurs 125
 Warnings 123
 Warnings Plus 124
 Messages d'erreur
 Erreurs attachées à une ligne précise 113
 Erreurs générales 112, 131
 Opérateurs 130
 Paramètres 129
 Syntaxe 127
 Typage 125
 Types 112
 Warnings 114

Méthodes

Liste 111
 Optimisation 32
 Modifiers (variable système) 60
 MouseDown (variable système) 60
 MouseProc (variable système) 60
 MouseX (variable système) 60
 MouseY (variable système) 60
 Multi-plate-forme 50

N

Nom de la base compilée 86, 88
 Notions de base 23
 Nouveau (menu Fichier) 82
 Numériques
 Optimisation 76
 Type par défaut 99
 Numérotation automatique des versions 100

O

Objets compilés 13
 OK (variable système) 60
 OLE (Windows) 53
 Optimisation
 des méthodes 32
 du code 75
 Option 96
 Options de compilation
 Caractéristiques générales 101
 Fenêtres 87
 Ouvrir (menu Fichier) 83

P

Paramètres
 Indirections 58
 Passation 56
 PAS DE TRACE 71
 Passation des paramètres 56
 Passe de typage 97
 Conclusion 105
 directif 104
 variables locales 104
 variables process et interprocess 104
 Personnaliser
 4D Compiler 139
 l'application 135

- l'icône de l'application 136, 137
- la signature de l'application 135
- les icônes des fichiers annexes 136
- Plug-Ins
 - Compilation avec des 50
 - Présentation 49
- Pointeur vers 69
- Pointeurs
 - Moyen d'éviter les retypages 56
 - Optimisation 80
 - Utilisation dans les routines de tableaux 68
- Privé (attribut d'objet de composant) 56
- Processus de compilation 102
- Projet par défaut (menu Fichier) 85
- Projets
 - Ouvrir 84
 - Présentation 83
- Protection des applications 16
- Protégé (attribut d'objet de composant) 56
- Public (attribut d'objet de composant) 56

Q

- Quitter (menu Fichier) 86

R

- RecDelimit (variable système) 60
- Recompiler (menu Fichier) 85
- Retypages
 - implicites 44
 - Utilisation des pointeurs 56
- rgrg (valeur aberrante) 97
- Routines externes
 - recevant des paramètres implicites 54

S

- Script Manager 94
- SELECTION VERS TABLEAU 67
- Signature de l'application (modifier) 135
- STOP 65
- Stop 106
 - compilation 103
- Suite... (bouton) 87
- Suspendre 106
 - compilation 103
- Symboles
 - Table des 92, 109

- Syntaxe
 - Précisions 61

T

- Table des symboles 92, 109
 - Création 25
- TABLEAU VERS ENUMERATION 68
- TABLEAU VERS SELECTION 67
- Tableaux
 - à deux dimensions (optimisation) 79
 - chaînes fixes 44
 - Changer le nombre de dimensions 43
 - Changer le type des éléments 43
 - locaux 68
 - Pointeurs dans les routines 68
- Tableaux (précisions de syntaxe) 66
- TRACE 71
- Traduction (Customizer Plus) 140
- Typage
 - Fichier 97, 119
 - Passes 97
- Typage des variables 24
 - dessinées dans les formulaires 45
 - par 4D Compiler 26, 33
 - par le développeur 33
 - Phase de compilation 103
- Type 70
- Type des alphanumériques 100
- Type des boutons par défaut 98
- Type des éléments d'un tableau
 - changer 43
- Type des numériques par défaut 99
- Type des variables
 - Cas d'ambiguïté 29
 - Description 24

U

- Utilisation d'une base compilée 107

V

- VALEURS DISTINCTES 67
- Variables
 - Conflits entre deux utilisations 39
 - créées par des routines externes 55
 - Déduction des types 26
 - des états rapides 60

- Nom 23
 - réservées 60
 - système 60
- Typage 24
 - Typage par 4D Compiler 26
- Types (rappel) 24
- Variables (précisions de syntaxe) 71
- Variables dessinées
 - considérées comme des Numériques 45
 - considérées comme du Texte 47
 - Zone externe 46
- Variables locales 80
 - \$0,... \$N 56
 - Conflits 42
 - Initialisation 96
 - Liste 111
 - Optimisation 80
 - Valeur aberrante 97
- Variables process et interprocess
 - Liste 110
- Variables réservées 60
- Variables système 60
- Version précédente (menu Fichier) 85

W

- Warnings 94, 114, 123
- Warnings Plus 124
- WIN4DX 49
- Windows
 - Compilation multi-plate-forme 50
- Windows (version) 10

