Guide du langage Pascal Objet

Borland®

Pascal Objet

Les applications mentionnées dans ce manuel sont brevetées ou en attente de brevet. Ce document ne donne aucun droit sur ces brevets.

COPYRIGHT © 1983, 2001 Borland Software Corporation. Tous droits réservés. Tous les produits Borland sont des marques ou des marques déposées de Borland Software Corporation. Tous les autres noms de produits sont des marques déposées de leurs fabricants respectifs.

Imprimé en Irlande

ALP0000WW21000 1E0R0201
0102030405-9 8 7 6 5 4 3 2 1
D3

Table des matières

Chapitre 1
Introduction 1-1

Que contient ce manuel ? 1-1
- Utilisation de Pascal Objet 1-1
- Conventions relatives à la documentation . 1-2

Autres sources d'information 1-3
Enregistrement logiciel et support technique . 1-3

Partie I
Descriptions des principes de base du langage

Chapitre 2
Présentation 2-1

Organisation d'un programme. 2-1
- Fichiers source Pascal 2-2
- Autres fichiers utilisés pour générer des applications. 2-2
- Fichiers générés par le compilateur 2-3

Programmes exemple 2-3
- Une application console simple 2-4
- Un exemple plus sophistiqué. 2-5
- Une application native 2-6

Chapitre 3
Programmes et unités 3-1

Structure et syntaxe d'un programme 3-1
- En-tête de programme. 3-2
- Clause uses d'un programme 3-3
- Le bloc. 3-3

Structure et syntaxe d'unité 3-3
- En-tête d'unité 3-4
- Section interface 3-4
- Section implémentation 3-4
- Section initialisation 3-5
- Section finalisation 3-5

Références d'unité et la clause uses 3-5
- Syntaxe de la clause uses 3-6
- Références d'unité multiples et indirectes . 3-7
- Références d'unité circulaires. 3-8

Chapitre 4
Eléments syntaxiques 4-1

Eléments syntaxiques fondamentaux 4-1
- Symboles spéciaux 4-2
- Identificateurs 4-2
  - Identificateurs qualifiés 4-3
- Mots réservés 4-3
- Directives. 4-4
- Nombres 4-4
- Labels 4-4
- Chaînes de caractères. 4-5

Commentaires et directives de compilation. . . 4-6
Expressions 4-6
- Opérateurs 4-6
  - Opérateurs arithmétiques. 4-7
  - Opérateurs booléens 4-8
  - Opérateurs logiques bit-à-bit. 4-9
  - Opérateurs de chaînes 4-10
  - Opérateurs de pointeurs 4-10
  - Opérateurs d'ensembles 4-11
  - Opérateurs relationnels 4-12
  - Opérateurs de classes 4-13
  - L'opérateur @. 4-13
  - Règles de priorité des opérateurs 4-14
- Appels de fonctions. 4-15
- Constructeurs d'ensembles 4-15
- Indices 4-16
- Transtypage 4-16
  - Transtypage de valeur 4-16
  - Transtypage de variable 4-17

Déclarations et instructions 4-18
- Déclarations 4-18
- Instructions. 4-19
- Instructions simples. 4-19
  - Instructions d'affectation 4-20
  - Appels de procédures et de fonctions . 4-20
  - Instructions goto 4-21
- Instructions structurées. 4-22
  - Instructions composées 4-22
  - Instructions With 4-23
  - Instructions If 4-24
  - Instructions Case 4-26

Boucles de contrôle 4-27
Instructions repeat 4-28
Instructions while 4-28
Instructions for 4-29
Blocs et portée 4-30
Blocs 4-30
Portée 4-31
Conflits de nom 4-32

Chapitre 5
Types de données, variables et constantes 5-1

A propos des types 5-1
Types simples 5-3
Types scalaires 5-3
Types entiers 5-4
Types caractère 5-5
Types booléens 5-6
Types énumérés 5-6
Types intervalle 5-8
Types réels 5-10
Types chaîne 5-11
Chaînes courtes 5-12
Chaînes longues 5-13
Chaînes étendues 5-13
A propos des jeux de caractères étendus 5-14
Manipulation des chaînes à zéro terminal . 5-14
Utilisation des pointeurs, tableaux et des constantes chaînes 5-15
Mélange de chaînes Pascal et de chaînes à zéro terminal 5-16
Types structurés 5-18
Ensembles 5-18
Tableaux 5-19
Tableaux statiques 5-19
Tableaux dynamiques 5-20
Types tableau et affectations 5-23
Enregistrements 5-23
Partie variable d'enregistrements 5-24
Types fichier 5-26
Pointeurs et types pointeur 5-27
Présentation des pointeurs 5-28
Types pointeur 5-29
Pointeurs de caractère 5-30
Autres types standard de pointeurs . . . 5-30
Types procédure 5-30
Types procédure dans les instructions et les expressions 5-32
Types variants 5-33
Conversions de types variants 5-34
Utilisation de variants dans les expressions 5-36
Tableaux variants 5-36
OleVariant 5-37
Compatibilité et identité de types 5-37
Identité de types 5-38
Compatibilité de types 5-38
Compatibilité pour l'affectation 5-39
Déclaration de types 5-40
Variables 5-41
Déclaration de variables 5-41
Adresses absolues 5-42
Variables dynamiques 5-42
Variables locales de thread 5-43
Constantes déclarées 5-43
Vraies constantes 5-44
Expressions constantes 5-45
Chaînes de ressource 5-45
Constantes typées 5-46
Constantes tableau 5-46
Constantes enregistrement 5-47
Constantes procédure 5-48
Constantes pointeur 5-48

Chapitre 6
Procédures et fonctions 6-1

Déclaration de procédures et de fonctions . . . 6-1
Déclaration de procédures 6-2
Déclaration de fonctions 6-3
Conventions d'appel 6-5
Déclaration forward et interface 6-6
Déclarations externes 6-7
Liaison de fichiers objet 6-7
Importation des fonctions de bibliothèques 6-7
Redéfinition de procédures et de fonctions 6-8
Déclarations locales 6-10
Routines imbriquées 6-10
Paramètres 6-10
Sémantique des paramètres 6-11
Paramètres valeur et paramètres variables 6-11
Paramètres constantes 6-13
Paramètres de sortie 6-13
Paramètres sans type 6-14
Paramètres chaîne 6-14

Paramètres tableau 6-15
Paramètres tableau ouvert. 6-15
Paramètres tableau ouvert variant 6-17
Paramètres par défaut 6-18
Paramètres par défaut et routines redéfinies. 6-19
Paramètres par défaut dans les déclarations forward et interface . .6-19
Appel de procédures et de fonctions 6-19
Constructeurs de tableaux ouverts 6-20

Chapitre 7
Classes et objets 7-1

Types classe. 7-2
Héritage et portée 7-3
TObject et TClass 7-3
Compatibilité des types classe 7-4
Types objet. 7-4
Visibilité des membres de classes 7-4
Membres privés, protégés et publics. . . 7-5
Membres publiés 7-5
Membres automatisés 7-6
Déclarations avancées et classes mutuellement dépendantes 7-7
Champs 7-8
Méthodes 7-9
Déclarations et implémentations des méthodes 7-9
Inherited 7-10
Self 7-10
Liaison de méthode 7-11
Méthodes statiques 7-11
Méthodes virtuelles et dynamiques . . .7-11
Méthodes abstraites 7-13
Redéfinition de méthodes. 7-13
Constructeurs. 7-14
Destructeurs 7-16
Méthodes de messages 7-16
Implémentation des méthodes de messages 7-17
Répartition des messages 7-18
Propriétés 7-18
Accès aux propriétés. 7-19
Propriétés tableau 7-21
Spécificateurs d'indice. 7-22
Spécificateurs de stockage 7-23
Surcharge et redéfinition de propriétés . . .7-24
Références de classe 7-25
Types de référence de classe. 7-25
Constructeurs et références de classe . . 7-26
Opérateurs de classe 7-27
Opérateur is 7-27
Opérateur as 7-27
Méthodes de classe 7-28
Exceptions 7-28
Quand utiliser des exceptions 7-29
Déclaration des types exception. 7-29
Déclenchement et gestion des exceptions . 7-30
Instructions Try...except. 7-31
Redéclenchement d'exceptions 7-33
Exceptions imbriquées 7-34
Instructions try...finally 7-34
Classes et routines standard des exceptions 7-35

Chapitre 8
Routines standard et Entrées/Sorties 8-1

Entrées et sorties de fichier 8-1
Fichier texte .8-3
Fichiers sans type 8-4
Pilotes de périphérique de fichiers texte 8-5
Fonctions de périphérique 8-6
Fonction Open 8-6
Fonction InOut. 8-6
Fonction Flush 8-6
Fonction Close 8-7
Gestion des chaînes à zéro terminal 8-7
Chaînes de caractères étendus. 8-8
Autres routines standard 8-8

Partie II
Sujets spéciaux

Chapitre 9
Bibliothèques et paquets 9-1

Appel de bibliothèques à chargement dynamique. .9-1
Chargement statique 9-2
Chargement dynamique 9-2
Ecriture de bibliothèques à chargement dynamique. .9-4
Clause exports 9-5

Code d'initialisation d'une bibliothèque . . 9-6
Variables globales d'une bibliothèque. . . . 9-7
Bibliothèques et variables système. 9-7
Exceptions et erreurs d'exécution dans les bibliothèques 9-8
Gestionnaire de mémoire partagée (Windows seulement) 9-8
Paquets 9-9
Déclarations et fichiers source de paquets . 9-9
Nom de paquets 9-10
Clause requires 9-11
Clause contains 9-11
Compilation de paquets. 9-12
Fichiers générés. 9-12
Directives de compilation spécifiques aux paquets 9-12
Options du compilateur ligne de commande spécifiques aux paquets 9-13

Chapitre 10
Interfaces d'objets 10-1

Types interface 10-1
IInterface et l'héritage 10-2
Identification d'interface 10-3
Conventions d'appel des interfaces 10-3
Propriétés d'interface 10-4
Déclarations avancées 10-4
Implémentation des interfaces 10-5
Clauses de résolution de méthode. 10-6
Modification de l'implémentation héritée .10-6
Implémentation des interfaces par délégation. 10-7
Délégation à une propriété de type interface 10-7
Délégation à une propriété de type classe 10-8
Références d'interface 10-9
Compatibilité des affectations d'interfaces 10-10
Transtypage d'interfaces 10-10
Interrogation d'interface 10-11
Objets automation (Windows seulement) . . 10-11
Types interface de répartition (Windows seulement) 10-11
Méthodes des interfaces de répartition (Windows seulement) 10-12
Propriétés des interfaces de répartition 10-12
Accès aux objets automation (Windows seulement). 10-12
Syntaxe des appels de méthode d'objet automation. 10-13
Interfaces doubles (Windows seulement) 10-14

Chapitre 11
Gestion de la mémoire 11-1

Le gestionnaire de mémoire (Windows seulement) 11-1
Variables 11-2
Formats de données internes 11-3
Types entiers. 11-3
Types caractères 11-3
Types booléens 11-3
Types énumérés 11-4
Types réels 11-4
Type Real48. 11-4
Type Single 11-4
Type Double 11-5
Type Extended 11-5
Type Comp 11-5
Type Currency 11-5
Types Pointer 11-5
Types chaînes courtes. 11-6
Types chaînes longues 11-6
Types chaînes étendues. 11-7
Types ensembles. 11-7
Types tableaux statiques 11-8
Types tableaux dynamiques 11-8
Types enregistrements 11-8
Types fichiers 11-9
Types procédures 11-11
Types classes. 11-11
Types références de classe 11-12
Types variants 11-12

Chapitre 12
Contrôle des programmes 12-1

Paramètres et résultats de fonction. 12-1
Transfert de paramètre 12-1
Règles d'enregistrement des registres. . 12-3
Résultats de fonction 12-3
Appels de méthode 12-4
Constructeurs et destructeurs 12-4
Procédures de sortie 12-5

Chapitre 13

Code assembleur en ligne 13-1

L'instruction asm 13-2
Utilisation des registres 13-2
Syntaxe des instructions assembleur 13-2
Labels 13-3
Codes opératoires d'instruction 13-3
Dimension de l'instruction RET 13-3
Dimension du saut automatique 13-3
Directives assembleur 13-4
Opérandes. 13-6
Expressions 13-6
Différences entre les expressions Pascal Objet et assembleur 13-7
Eléments d'expression 13-8
Constantes 13-8
Registres 13-9
Symboles 13-10
Classes d'expression 13-12
Types d'expression 13-13
Opérateurs d'expression 13-14
Procédures et fonctions assembleur 13-16

Annexe A

Grammaire du Pascal Objet A-1

Index I-1

Chapitre

1

Introduction

Ce manuel décrit le langage de programmation Pascal Objet tel qu'il est utilisé dans les outils de développement Borland.

Que contient ce manuel ?

Les sept premiers chapitres décrivent la plupart des éléments du langage utilisés couramment dans la programmation. Le chapitre 8 décrit brièvement les routines standard d'Entrées/Sorties de fichier et de manipulation de chaînes.

Les chapitres suivants décrivent les extensions et les restrictions du langage concernant les bibliothèques à chargement dynamique et les paquets (chapitre 9), et les interfaces d'objets (chapitre 10). Les trois derniers chapitres abordent des sujets plus techniques : la gestion mémoire (chapitre 11), le contrôle de programme (chapitre 12) et les routines en langage assembleur dans des programmes Pascal Objet (chapitre 13).

Utilisation de Pascal Objet

Le *Guide du langage Pascal Objet* décrit l'utilisation du langage Pascal Objet avec les systèmes d'exploitation Linux ou Windows. Les différences relatives aux plates-formes sont signalées chaue fois que c'est nécessaire.

La plupart des développeurs Delphi ou Kylix écrivent et compilent leurs programmes Pascal Objet dans l'environnement de développement intégré (EDI). L'utilisation de l'EDI permet au produit de gérer de nombreux détails de la configuration des projets et des fichiers sources, par exemple la maintenance des informations sur les dépendances entre les unités. Les produits Borland imposent par ailleurs des contraintes sur l'organisation des programmes qui ne font pas, à proprement parler, partie des spécifications du langage Pascal Objet. Ainsi, certaines conventions sur les noms de fichier ou de programme peuvent être

évitées si vous écrivez vos programmes en dehors de l'EDI et si vous les compilez depuis la ligne de commande.

Généralement, ce manuel suppose que l'utilisateur emploie l'EDI et conçoit des applications qui utilisent la bibliothèque de composants visuels (VCL) ou la bibliothèque de composants Borland multiplate-forme (CLX). Néanmoins, dans certains cas, les règles propres à Borland sont distinguées des règles s'appliquant à tous les programmes Pascal Objet.

Conventions relatives à la documentation

Les identificateurs (c'est-à-dire les noms de constantes, variables, types, champs, propriétés, procédures, fonctions, programmes, unités, bibliothèques ou paquets) apparaissent en *italique* dans le texte. Les opérateurs Pascal Objet, les mots réservés et les directives sont en **gras**. Le code exemple et le texte devant être saisi littéralement (dans un fichier ou sur la ligne de commande) utilisent une `police à pas fixe`.

Dans les listings de programme, les mots réservés et les directives apparaissent en **gras**, comme dans le texte :

```
function Calculate(X, Y: Integer): Integer;
begin
  ⋮
end;
```

C'est ainsi que l'éditeur de code affiche les mots réservés et les directives si l'option de mise en évidence syntaxique est activée.

Certains exemples de programme, comme le précédent, contiennent des points de suspension (... ou ⋮). Ces points de suspension représentent du code supplémentaire qui doit être ajouté dans un véritable fichier. Ils ne doivent pas être recopiés littéralement.

Dans les descriptions de syntaxe, l'*italique* indique l'élément auquel, dans le code réel, vous devez substituer des constructions syntaxiquement correctes. Par exemple, l'en-tête de déclaration de la fonction déclarée ci-dessus peut être représenté par :

`function` *nomFonction*`(`*ListeArguments*`):` *typeRenvoyé*`;`

Les descriptions de syntaxe contiennent également des points de suspension (...) et des indices :

`function` *nomFonction*`(`arg_1`, ...,` arg_n`):` *TypeRenvoyé*`;`

Autres sources d'information

Le système d'aide en ligne de votre outil de développement propose des informations sur l'EDI et l'interface utilisateur, ainsi que les données les plus récentes sur la VCL et/ou sur CLX. La plupart des sujets concernant la programmation, comme le développement de bases de données, sont abordés en détail dans le *Guide du développeur*. Pour une présentation générale de la documentation, voir le manuel *Prise en main* livré avec le produit.

Enregistrement logiciel et support technique

Borland Software Corporation offre aussi de nombreuses possibilités de support si vous avez besoin d'informations supplémentaires. Pour recevoir des informations sur ce produit, remplissez la carte d'enregistrement et renvoyez-la. Pour des informations sur les services de support développeur Borland, visitez le site Web à l'adresse www.borland.fr.

Partie

I

Descriptions des principes de base du langage

Les chapitres de la partie I présentent les éléments essentiels les plus souvent nécessaires à la programmation. Ces chapitres sont :

- Chapitre 2, "Présentation"
- Chapitre 3, "Programmes et unités"
- Chapitre 4, "Eléments syntaxiques"
- Chapitre 5, "Types de données, variables et constantes"
- Chapitre 6, "Procédures et fonctions"
- Chapitre 7, "Classes et objets"
- Chapitre 8, "Routines standard et Entrées/Sorties"

Chapitre

2

Présentation

Le Pascal Objet est un langage compilé de haut niveau à types stricts qui gère la conception structurée et orientée objet. Ces qualités sont la lisibilité du code, une compilation rapide et l'utilisation d'unités multiples permettant une programmation modulaire.

Le Pascal Objet propose des caractéristiques spéciales qui gèrent le modèle de composant et l'environnement RAD de Borland. Dans la plupart des cas, les descriptions et exemples de ce manuel supposent que vous utilisez le Pascal Objet pour développer des applications en utilisant des outils de développement Borland comme Delphi ou Kylix.

Organisation d'un programme

Les programmes sont généralement divisés en modules de code source appelés *unités*. Chaque programme commence par un en-tête qui spécifie le nom du programme. L'en-tête est suivi d'une clause **uses** facultative puis d'un bloc de déclarations et d'instructions. La clause **uses** énumère toutes les unités liées à ce programme ; ces unités, qui peuvent être partagées par différentes programmes, peuvent également avoir leur propre clause **uses**.

La clause **uses donne au compilateur des** informations sur les dépendances entre modules. Comme ces informations sont stockées dans les modules mêmes, les programmes Pascal Objet n'ont pas besoin de makefiles, de fichiers d'en-tête ou de directives "include" du préprocesseur. Le gestionnaire de projet génère un fichier makefile à chaque fois que le fichier est chargé dans l'EDI, mais il ne l'enregistre que pour les groupes de projets contenant plusieurs projets.

Pour davantage d'informations sur la structure d'un programme et les dépendances, voir chapitre 3, "Programmes et unités".

Fichiers source Pascal

Le compilateur s'attend à trouver du code source Pascal sous trois formes différentes :

- *des fichiers source d'unité* (se terminant par l'extension .pas)
- *des fichiers projet* (se terminant par l'extension .dpr)
- *des fichiers source de paquet* (se terminant par *l'extension* .dpk)

Les fichiers sources d'unité contiennent l'essentiel du code d'une application. Chaque application n'a qu'un seul fichier projet et plusieurs fichiers unité ; le fichier projet (qui correspond au fichier programme "principal" en Pascal standard) organise les fichiers unité en une application. Les outils de développement Borland gèrent automatiquement un fichier projet pour chaque application.

Si vous compilez un programme depuis la ligne de commande, vous pouvez placer tout le code source dans des fichiers unité (.pas). Mais si vous utilisez l'EDI pour concevoir votre application, vous devez avoir un fichier projet (.dpr).

Les fichiers source de paquet sont semblables au fichiers projet, mais ils sont utilisés pour créer des bibliothèques de liaison dynamique spéciales appelées des *paquet*s. Pour davantage d'informations sur les paquets, voir chapitre 9, "Bibliothèques et paquets".

Autres fichiers utilisés pour générer des applications

Outre les modules de code source, les produits Borland utilisent plusieurs fichiers ne contenant pas de code Pascal pour générer les applications. Ces fichiers sont gérés automatiquement, ce sont :

- *les fichiers fiche,* d'extension .dfm (Delphi) ou .xfm (Kylix),
- *les fichiers ressource,* d'extension .res et
- *les fichiers d'options de projet* d'extension .dof (Delphi) ou .xof (Kylix).

Un fichier fiche est soit un fichier texte, soit un fichier de ressources compilées qui peut contenir des bitmaps, des chaînes, etc. Chaque fichier fiche représente une seule fiche correspondant généralement à une fenêtre ou à une boîte de dialogue d'une application. L'EDI vous permet de visualiser et de modifier les fichiers fiche sous forme de texte et d'enregistrer les fichiers fiche en format texte ou en format binaire. Bien que par défaut les fichiers fiche soient enregistrés sous forme de texte, vous ne les modifierez généralement pas manuellement ; vous utiliserez plutot les outils de conception visuels Borland. Chaque projet contient au moins une fiche et chaque fiche dispose d'un fichier unité (.pas) associé portant, par défaut, le même nom que le fichier fiche.

Outre les fichiers fiche, chaque projet utilise un fichier de ressource (.res) standard pour contenir le bitmap de l'icône de l'application. Par défaut ce fichier porte le même nom que le fichier projet (.dpr). Pour changer l'icône d'une application, utilisez la boîte de dialogue Options de projet.

Un fichier d'options de projet (.dof ou .kof) contient les paramètres du compilateur et du lieur, les répertoires de recherche, les informations de version, etc. Chaque projet a un fichier d'options de projet associé qui porte le même nom que le fichier projet (.dpr). Ces options sont généralement définies dans la boîte de dialogue Options de projet.

Divers outils de l'EDI permettent de stocker les données dans des fichiers de types différents. Les fichiers de configuration du bureau (.dsk) contiennent des informations sur la disposition des fenêtres et d'autres options de configuration ; les fichiers de configuration du bureau peuvent être spécifiques au projet ou concerner l'environnement tout entier. Ces fichiers n'ont pas d'effet direct sur la compilation.

Fichiers générés par le compilateur

Lors de la première génération d'une application ou une bibliothèque de liaison dynamique standard, le compilateur produit un fichier unité compilée .dcu (Windows) ou .dcu/.dpu (Linux) pour chaque nouvelle unité utilisée dans le projet ; tous les fichiers .dcu (Windows) ou .dcu/.dpu (Linux) du projet sont ensuite liés pour créer un seul fichier exécutable ou une seule bibliothèque partagée. A la première génération d'un paquet, le compilateur produit un fichier .dcu (Windows) ou .dpu (Linux) pour chaque nouvelle unité contenue dans le paquet puis il crée à la fois un fichier .dcp et un fichier paquet. Pour davantage d'informations sur les bibliothèques et les paquets, voir chapitre 9, "Bibliothèques et paquets". Si vous utilisez l'option **–GD**, le lieur génère également un fichier map et un fichier .drc ; le fichier .drc qui contient des chaînes de ressource peut être compilé en un fichier ressource.

Quand vous régénérez un projet, les unités individuelles ne sont pas recompilées sauf si leur fichier source (.pas) a été modifié depuis la dernière compilation, si leurs fichiers .dcu (Windows) ou .dcu/.dpu (Linux) ne peuvent être trouvés ou si vous demandez explicitement au compilateur de les régénérer. En fait, il n'est même pas nécessaire que le fichier source d'une unité soit disponible du moment que le compilateur trouve le fichier unité compilée.

Programmes exemple

Les exemples suivants décrivent les caractéristiques de base de la programmation Pascal Objet. Les exemples montrent de simples applications Pascal Objet qui ne peuvent pas être compilées depuis l'EDI ; mais vous pouvez les compiler depuis l'invite de commande.

Une application console simple

Le programme suivant est une application console simple que vous pouvez compiler et exécuter depuis l'invite de commande.

```
program Greeting;

{$APPTYPE CONSOLE}

var MyMessage: string;

begin
  MyMessage := 'Bonjour chez vous!';
  Writeln(MyMessage);
end.
```

La première ligne déclare un programme appelé *Greeting*. La directive `{$APPTYPE CONSOLE}` indique au compilateur que c'est une application console qui doit être exécutée depuis la ligne de commande. La ligne suivante déclare une variable appelée *MyMessage*, qui contient une chaîne. Pascal objet dispose directement d'un type de donnée chaîne. Puis le programme affecte la chaîne "Bonjour chez vous!" à la variable *MyMessage* et envoie le contenu de *MyMessage sur la sortie* standard en utilisant *la procédure Writeln*. La procédure *Writeln* est définie implicitement dans l'unité *System* qui est incluse automatiquement dans toutes les applications.

Vous pouvez saisir ce programme dans un fichier appelé Greeting.pas ou Greeting.dpr, puis le compiler en entrant :

```
Dans Delphi: DCC32 Greeting
```

```
Dans Kylix: dcc Greeting
```

sur la ligne de commande. L'exécutable résultant affiche le message "Bonjour chez vous!"

Mis à part sa simplicité, cet exemple diffère largement du type de programme que vous allez probablement écrire avec les outils de développement Borland. Tout d'abord, c'est une application console. Les outils de développement Borland sont généralement utilisés pour écrire des applications ayant une interface graphique ; ainsi, vous ne devez normalement pas appeler *Writeln*. De plus, la totalité du programme exemple (à l'exception de *Writeln*) se trouve dans un seul fichier. Dans une application typique, l'en-tête du programme (la première ligne de cet exemple) se trouve dans un fichier projet séparé qui ne contient pas la logique réelle de l'application sinon quelques appels aux *méthodes* définies dans les fichiers unité.

Un exemple plus sophistiqué

L'exemple suivant propose un programme constitué de deux fichiers : un fichier *projet* et un fichier *unité*. Le fichier projet, que vous pouvez enregistrer sous le nom Greeting.dpr, a la forme suivante :

```
program Greeting;

{$APPTYPE CONSOLE}

uses Unit1;

begin
  PrintMessage('Bonjour chez vous!');
end.
```

La première ligne déclare un programme appelé *Greeting* qui, encore une fois, est une application console. La clause `uses Unit1;` spécifie au compilateur que *Greeting* inclut une unité appelée *Unit1*. Enfin le programme appelle la procédure *PrintMessage* en lui transmettant la chaîne "Bonjour chez vous!". Mais où se trouve la procédure *PrintMessage* ? Elle est définie dans *Unit1*. Voici le code source de *Unit1* que vous devez enregistrer dans un fichier appelé Unit1.pas :

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation

procedure PrintMessage(msg: string);
begin
  Writeln(msg);
end;

end.
```

Unit1 définit une procédure appelée *PrintMessage* qui attend un seul paramètre chaîne comme argument et envoie la chaîne dans la sortie standard. En Pascal, les routines qui ne renvoient pas de valeur sont appelées des *procédures*. Les routines qui renvoient une valeur sont appelées des *fonctions*. Remarquez la double déclaration de *PrintMessage dans Unit1*. La première déclaration, après le mot réservé **interface**, rend *PrintMessage* accessible aux modules (comme *Greeting*) qui utilisent *Unit1*. Le seconde déclaration, placée après le mot réservé **implementation**, définit réellement *PrintMessage*.

Vous pouvez maintenant compiler *Greeting* depuis la ligne de commande en saisissant :

```
Dans Delphi: DCC32 Greeting

Dans Kylix: dcc Greeting
```

Il n'est pas nécessaire d'inclure *Unit1* comme argument de la ligne de commande. Quand le compilateur traite le fichier Greeting.dpr, il recherche automatiquement les fichiers unité dont dépend le programme *Greeting*. L'exécutable résultant fait la même chose que le premier exemple : il affiche le message "Bonjour chez vous!"

Une application native

L'exemple suivant est une application construite en utilisant les composants VCL ou CLX de l'EDI. Ce programme utilisant des fichiers fiche et ressource générés automatiquement, il n'est pas possible de compiler le code source tout seul. Mais ce programme illustre des caractéristiques importantes en Pascal Objet. Outre diverses unités, le programme utilise classes et objets, qui sont décrits en détail dans le chapitre 7, "Classes et objets".

Ce programme contient un fichier projet et deux nouveaux fichiers unité. Tout d'abord le fichier projet :

```
program Greeting;  { les commentaires sont entre accolades }

uses
  Forms, {changer le nom d'unité en QForms pour Linux}
  Unit1 in 'Unit1.pas' { l'unité de Form1 },
  Unit2 in 'Unit2.pas' { l'unité de Form2 };

{$R *.res}  { Cette directive lie le fichier ressource du projet }

begin
  { Appel de Application }
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Là encore, le programme s'appelle *Greeting*. Il utilise trois unités : *Forms*, qui fait partie de VCL et de CLX ; *Unit1* qui est associée à la fiche principale de l'application (*Form1*) et *Unit2* qui est associée à une autre fiche (*Form2*).

Le programme fait une série d'appels à un objet nommé *Application* qui est une instance de la classe *TApplication* définie dans l'unité *Forms*. Chaque projet dispose d'un objet *Application* généré automatiquement. Deux de ces appels invoquent une méthode de *TApplication appelée CreateForm*. Le premier appel de *CreateForm* crée *Form1*, une instance de la classe *TForm1* définie dans *Unit1*. Le deuxième appel de *CreateForm* crée *Form2*, une instance de la classe *TForm2* définie dans *Unit2*.

Unit1 a la forme suivante :

```
unit Unit1;

interface

uses  { ces unités font partie de la bibliothèque VCL }
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

{
Dans Linux, la clause uses se présente comme ceci :
uses  { ces unités font partie de CLX }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

uses Unit2;  { C'est là que Form2 est défini}

{$R *.dfm}  { cette directive lie le fichier fiche de Unit1 }

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;
end;

end.
```

Unit1 crée une classe nommée *TForm1* (dérivée de la classe *TForm*) et une instance de cette classe, *Form1*. *TForm1* contient un bouton (*Button1*, une instance de *TButton*) et une procédure nommée *TForm1.Button1Click* qui est appelée lors de l'exécution à chaque fois que l'utilisateur clique sur *Button1*. *TForm1.Button1Click* cache *Form1* et affiche *Form2* (l'appel *Form2.ShowModal*). *Form2* est défini dans *Unit2* :

```
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
```

```
{
Dans Linux, la clause uses se présente comme ceci :
uses  { ces unités font partie de CLX }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}
type
  TForm2 = class(TForm)
    Label1: TLabel;
    CancelButton: TButton;
    procedure CancelButtonClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  end;

var
  Form2: TForm2;

implementation

uses Unit1;

{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
  Form2.Close;
end;

end.
```

Unit2 crée une classe nommée *TForm2* et une instance de cette classe, *Form2*. *TForm2* contient un bouton (*CancelButton*, une instance de *TButton*) et un libellé (*Label1*, une instance de *TLabel*). Vous ne pouvez le voir dans le code source, mais *Label1* affiche un intitulé contenant le texte “Bonjour chez vous!”. L’intitulé est défini dans le fichier fiche de *Form2*, Unit2.dfm.

Unit2 définit une procédure. *TForm2.CancelButtonClick* est appelée lors de l’exécution à chaque fois que l’utilisateur clique sur *CancelButton* ; elle ferme *Form2*. Cette procédure (ainsi que *TForm1.Button1Click de Unit1*) est appelée *gestionnaire d’événement* car elle répond à des événements se produisant lors de l’exécution du programme. Les gestionnaires d’événements sont attribués à des événements spécifiques par les fichiers fiche (.dfm dans Windows, .xfm dans Linux) de *Form1* et de *Form2*.

Au démarrage du programme *Greeting, la fiche Form1* est affichée et *Form2* est invisible. Par défaut, seule la première fiche créée dans le fichier projet est visible à l’exécution. C’est ce qu’on appelle la *fiche principale du projet*. Quand l’utilisateur choisit le bouton placé dans *Form1*, *Form1* disparaît et est remplacée par *Form2*, qui affiche la salutation “Bonjour chez vous!”. Quand l’utilisateur ferme la fiche *Form2* (en choisissant le bouton *Annuler* ou le bouton de fermeture dans la barre de titre), *Form1* réapparaît.

Chapitre

3

Programmes et unités

Un programme est construit à partir de modules de code source appelés des *unités*. Chaque unité est stockée dans un fichier distinct et compilée séparément ; les unités compilées sont liées pour créer une application. Les unités permettent :

- De diviser de grands programmes en modules qui peuvent être modifiés séparément.
- De créer des bibliothèques qui peuvent être partagées par plusieurs programmes.
- De distribuer des bibliothèques à d'autres développeurs sans leur donner le code source.

En programmation Pascal classique, tout le code source, y compris le programme principal, est placé dans des fichiers .pas. Les outils Borland utilisent un fichier *projet* (.dpr) pour stocker le programme "principal" alors que tout le reste du code source est placé dans des fichiers *unité* (.pas). Chaque application (ou *projet*) est constitué d'un seul fichier projet et d'un ou de plusieurs fichiers unité. Strictement parlant, il n'est pas nécessaire d'utiliser explicitement des unités dans un projet mais tous les programmes utilisent implicitement l'unité *System*. Pour générer un projet, le compilateur doit disposer, pour chaque unité, soit du fichier source soit du fichier unité compilé.

Structure et syntaxe d'un programme

Un programme contient :

- Un en-tête de programme ;
- une clause **uses** (facultative) ;
- un bloc de déclarations et d'instructions.

L'en-tête du programme spécifie le nom du programme. La clause **uses** énumère les unités utilisées par le programme. Le bloc contient les déclarations et les instructions exécutées au démarrage du programme. L'EDI s'attend à trouver ces trois éléments dans un seul fichier projet (.dpr).

L'exemple suivant est le fichier projet d'un programme appelé Editor.

```
1      program Editor;
2
3      uses
4        Forms, {changer en QForms pour Linux}
5        REAbout in 'REAbout.pas' {AboutBox},
6        REMain in 'REMain.pas' {MainForm};
7
8      {$R *.res}
9
10     begin
11       Application.Title := 'Editeur de texte';
12       Application.CreateForm(TMainForm, MainForm);
13       Application.Run;
14     end.
```

La ligne 1 contient l'en-tête du programme. La clause **uses** va des lignes 3 à 6. La ligne 8 est une directive de compilation qui lie au programme le fichier ressource du projet. Les lignes 10 à 14 contiennent le bloc d'instructions exécutées au lancement du programme. Enfin, le fichier projet, comme tous les fichiers source, se termine par un point.

Le code précédent est en fait un exemple assez standard de fichier projet. Les fichiers projet sont généralement courts car l'essentiel de la logique du programme se trouve dans les fichiers unité. Les fichiers projet sont générés et gérés automatiquement : il est donc rarement nécessaire de les modifier manuellement.

En-tête de programme

L'en-tête de programme spécifie le nom du programme. Il est composé du mot réservé **program** suivi d'un identificateur valide puis un point-virgule. L'identificateur doit correspondre au nom du fichier projet. Dans l'exemple ci-dessus, comme le programme s'appelle Editor, le fichier projet doit être appelé EDITOR.dpr.

En Pascal standard, un en-tête de programme peut contenir des paramètres qui suivent le nom du programme :

```
program Calc(input, output);
```

Le compilateur Pascal Objet de Borland ne tient pas compte de ces paramètres.

Clause uses d'un programme

La clause **uses** énumère les unités incorporées dans le programme. Ces unités peuvent, à leur tour, avoir leur propre clause **uses**. Pour davantage d'informations sur la clause **uses**, voir "Références d'unité et la clause uses" à la page 3-5.

Le bloc

Le bloc est constitué d'une instruction simple ou structurée qui est exécutée au lancement du programme. Dans la plupart des programmes, le bloc est constitué d'une instruction composite (encadrée par les mots réservés **begin** et **end**) contenant des instructions qui sont de simples appels aux méthodes de l'objet *Application* du projet. Chaque projet contient une variable *Application* qui contient une instance de *TApplication*, de *TWebApplication* ou de *TServiceApplication*. Le bloc peut également contenir des déclarations de constantes, types, variables, procédures ou de fonctions ; ces déclarations doivent précéder la partie instruction du bloc.

Structure et syntaxe d'unité

Une unité est constituée de types (y compris de classes), de constantes, de variables, et de routines (fonctions et procédures). Chaque unité est définie dans son propre fichier unité (.pas).

Un fichier unité commence par l'en-tête d'unité, suivi par les sections *interface*, *implémentation*, *initialisation* et *finalisation*. Les sections d'initialisation et de finalisation sont facultatives. Le squelette d'un fichier d'unité a la forme suivante :

```
unit Unit1;

interface

uses  { La liste des unités vient ici }

  { La section interface vient ici}

implementation

uses  { La liste des unités vient ici }

  { La section implémentation vient ici }

initialization
  { La section initialisation vient ici }

finalization
  { La section finalisation vient ici}

end.
```

L'unité doit se terminer par le mot réservé **end** suivi d'un point.

En-tête d'unité

L'en-tête d'unité spécifie le nom de l'unité. Il est constitué du mot réservé **unit** suivi d'un identificateur valide et d'un point-virgule. Pour les applications développées à l'aide d'outils Borland, l'identificateur doit correspondre au nom du fichier unité. Ainsi, l'en-tête d'unité :

```
unit MainForm;
```

doit se trouver dans un fichier source appelé MAINFORM.pas et le fichier contenant l'unité compilée se nomme MAINFORM.dcu.

Les noms d'unité doivent être unique à l'intérieur d'un projet. Même si les fichiers unité sont dans des répertoires différents, deux unités portant le même nom ne peuvent être utilisées dans un même programme.

Section interface

La section interface d'une unité commence par le mot réservé **interface** et se poursuit jusqu'à la section implémentation. La section interface déclare les constantes, types, variables, procédures et fonctions accessibles aux *clients*, c'est-à-dire aux unités ou programmes qui utilisent l'unité. Ces entités sont dite *publiques* car un client peut y accéder comme si elles étaient déclarées dans le client même.

La déclaration d'interface d'une procédure ou d'une fonction ne contient que l'en-tête de la routine. Le bloc de la procédure ou de la fonction se trouve dans la section implémentation. Ainsi les déclarations de procédure ou de fonction de la section interface fonctionnent comme des déclarations **forward** même si la directive **forward** n'est pas utilisée.

La déclaration d'interface d'une classe doit contenir la déclaration de tous les membres de la classe.

La section interface peut contenir sa propre clause **uses** qui doit apparaître immédiatement après le mot **interface**. Pour des informations sur la clause **uses**, voir “Références d'unité et la clause uses” à la page 3-5.

Section implémentation

La section implémentation d'une unité commence par le mot réservé **implementation** et se poursuit jusqu'au début de la section initialisation ou, en son absence, jusqu'à la fin de l'unité. La section implémentation définit les procédures et fonctions déclarées dans la section interface. A l'intérieur de la section implémentation, ces procédures et fonctions peuvent être définies et appelées dans un ordre quelconque. Vous pouvez omettre la liste des paramètres des en-têtes de procédures et fonctions publiques quand vous les définissez dans la section implémentation. Si vous spécifiez la liste des paramètres, elle doit correspondre exactement à la déclaration effectuée dans la section interface.

Outre la définition des procédures et fonctions publiques, la section implémentation peut déclarer des constantes, des types (y compris des classes), des variables, des procédures et des fonctions *privés* de l'unité (c'est-à-dire inaccessibles aux clients).

La section implémentation peut contenir sa propre clause **uses** qui doit apparaître immédiatement après le mot **implementation**. Pour des informations sur la clause **uses**, voir "Références d'unité et la clause uses" à la page 3-5.

Section initialisation

La section initialisation est facultative. Elle commence par le mot réservé **initialization** et se poursuit jusqu'au début de la section finalisation ou, en son absence, jusqu'à la fin de l'unité. La section initialisation contient des instructions qui sont exécutées, dans l'ordre où elles apparaissent, au démarrage du programme. Si, par exemple, vous avez défini des structures de données devant être initialisées, vous pouvez le faire dans la section initialisation.

La section initialisation des unités utilisées par un client sont exécutées dans l'ordre de leur énumération dans la clause **uses** du client.

Section finalisation

La section facultative de finalisation ne peut apparaître que dans les unités ayant une section initialisation. La section finalisation commence par le mot réservé **finalization** et se poursuit jusqu'à la fin de l'unité. Elle contient des instructions qui sont exécutées lors de l'arrêt du programme principal. Utilisez la section finalisation pour libérer des ressources allouées dans la section initialisation.

Les sections finalisation sont exécutées dans l'ordre inverse de celui de leur initialisation. Si, par exemple, votre application initialise, dans cet ordre, les unités *A*, *B* et *C* ; elle les finalise dans l'ordre *C*, *B* et *A*.

Une fois que le code d'initialisation d'une unité a commencé à s'exécuter, la section de finalisation correspondante s'exécute obligatoirement à l'arrêt de l'application. La section finalisation doit donc être capable de gérer des données dont l'initialisation a été incomplète. En effet, si une erreur d'exécution se produit, il est possible que la section initialisation ne soit pas complètement exécutée.

Références d'unité et la clause uses

Une clause **uses** énumère les unités utilisées par le programme, la bibliothèque ou l'unité où elle apparaît. Pour des informations sur les bibliothèques, voir chapitre 9, "Bibliothèques et paquets".) Une clause **uses** peut apparaître dans :

- le fichier projet d'un programme ou d'une bibliothèque ;
- la section interface d'une unité ;
- la section implémentation d'une unité.

La plupart des fichiers projet contiennent une clause **uses** tout comme la section interface de la plupart des unités. La section implémentation d'une unité peut également contenir sa propre clause **uses**.

L'unité *System* est utilisée automatiquement par toutes les applications et ne peut être spécifiée explicitement dans la clause **uses**. *System* implémente les routines pour les E/S de fichier, la manipulation de chaîne, les opérations à virgule flottante, l'allocation dynamique de mémoire, etc. D'autres unités de bibliothèque standard, comme *SysUtils*, peuvent être placées dans la clause **uses**. Dans la plupart des cas, toutes les unités nécessaires sont placées dans la clause **uses** quand votre projet génère et gère un fichier source.

Pour davantage d'informations sur l'emplacement et le contenu de la clause **uses**, voir "Références d'unité multiples et indirectes" à la page 3-7 et "Références d'unité circulaires" à la page 3-8.

Syntaxe de la clause uses

Une clause **uses** est constituée du mot réservé **uses**, suivi d'un ou de plusieurs noms d'unité séparés par des virgules et se termine par un point-virgule.

Exemples :

```
uses Forms, Main;

uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

Dans la clause **uses d'un** programme ou d'une unité, chaque nom d'unité peut être suivi du mot réservé **in** puis, entre guillemets, le nom d'un fichier source avec ou sans chemin d'accès ; le chemin d'accès peut être absolu ou relatif. Exemples :

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;

uses
  QForms,
  Main,
  Extra in '../extra/extra.pas';
```

Utilisez **in** ... après un nom d'unité si vous avez besoin de spécifier le nom du fichier source de l'unité. Comme l'EDI suppose que le nom de l'unité correspond au nom du fichier source dans lequel elle se trouve, il n'est généralement pas nécessaire d'utiliser cette option. Il n'est nécessaire d'utiliser **in** que si l'emplacement du fichier source est ambigu, par exemple si :

- Vous avez utilisé un fichier source placé dans un répertoire différent de celui du fichier projet et si ce répertoire n'est pas dans le chemin de recherche du compilateur, ni dans le chemin de recherche des bibliothèques global.
- Différents répertoires du chemin de recherche du compilateur contiennent des unités portant le même nom.

- Vous compilez une application console depuis la ligne de commande et le nom (identificateur) attribué à l'unité ne correspond pas à celui du fichier source.

Le compilateur se base aussi sur la construction **in** ... pour déterminer les unités qui font partie d'un projet. Seules les unités qui apparaissent dans la clause **uses** d'un fichier projet (.dpr) suivies de **in** et d'un nom de fichier sont considérées comme faisant partie du projet ; les autres unités de la clause **uses** sont utilisées par le projet mais ne lui appartiennent pas. Cette distinction n'a pas d'effet sur la compilation, mais affecte les outils de l'EDI comme le gestionnaire de projet et l'explorateur de projet.

Dans la clause **uses** d'une unité, vous ne pouvez utiliser **in** pour indiquer au compilateur où se trouve un fichier source. Chaque unité doit se trouver dans le chemin de recherche du compilateur, le chemin de recherche des bibliothèques global ou dans le même répertoire que l'unité qui l'utilise. De plus, le nom d'unité doit correspondre au nom du fichier source.

Références d'unité multiples et indirectes

L'ordre d'énumération des unités dans la clause **uses** détermine l'ordre de leur initialisation (voir "Section initialisation" à la page 3-5) et affecte la manière dont les identificateurs sont trouvés par le compilateur. Si deux unités déclarent une variable, une constante, un type, une procédure ou une fonction portant le même nom, le compilateur utilise l'élément défini dans l'unité énumérée en dernier dans la clause **uses**. Pour accéder à l'identificateur de l'autre unité, il faut ajouter un qualificateur : *UnitName.Identifier*.)

La clause **uses** ne doit inclure que les unités utilisées directement par le programme ou l'unité dans lequel la clause apparaît. Par exemple, si l'unité *A* fait référence à des constantes, types, variables, procédures ou fonctions déclarées dans l'unité *B*, alors *A* doit utiliser *B* explicitement. Si *B* fait, à son tour, référence à des identificateurs de l'unité *C*, alors *A* dépend indirectement de *C* ; dans ce cas, il n'est pas nécessaire de placer *C* dans la clause **uses** de *A*, mais le compilateur doit néanmoins pouvoir trouver *B et C* pour pouvoir traiter *A*.

L'exemple suivant illustre le phénomène de dépendance indirecte :

```
program Prog;
uses Unit2;
const a = b;
⋮
unit Unit2;
interface
uses Unit1;
const b = c;
⋮
unit Unit1;
interface
const c = 1;
⋮
```

Dans cet exemple, *Prog* dépend directement de *Unit2*, qui à son tour dépend directement de *Unit1*. De ce cas, *Prog* dépend indirectement de *Unit1*. Comme *Unit1* n'apparaît pas dans la clause **uses** de *Prog*, les identificateurs déclarés dans *Unit1* ne sont pas utilisables dans *Prog*.

Pour compiler un module client, le compilateur a besoin de trouver toutes les unités dont dépend, directement ou indirectement, le client. Si le code source de ces unités n'a pas été modifié, le compilateur n'a besoin que de leur fichier .dcu (Windows) ou .dcu/.dpu (Linux) et non de leur fichier source Pascal (.pas).

Quand vous avez apporté des modifications à la section interface d'une unité, les autres unités qui dépendent de celle-ci doivent être recompilées. Mais si la modification n'est faite que dans la partie implémentation ou dans une autre section de l'unité, il n'est pas nécessaire de recompiler les unités dépendantes. Le compilateur gère automatiquement ces dépendances et ne recompile les unités que lorsque cela est nécessaire.

Références d'unité circulaires

Quand des unités, directement ou pas, se font mutuellement référence, les unités sont dites mutuellement dépendantes. Les dépendances mutuelles sont autorisées tant qu'il n'y a pas création d'une référence circulaire reliant la clause **uses** de la section interface d'une unité à la clause **uses** d'une autre unité. En d'autres termes, partant de la section interface d'une unité, il ne faut pas pouvoir revenir à cette unité en suivant les dépendances des sections interface des autres unités. Une configuration de dépendance mutuelle est possible si chaque référence circulaire aboutit à la clause **uses** d'au moins une section implémentation.

Dans le cas le plus simple de deux unités mutuellement dépendantes, cela signifie que les unités ne peuvent se référencer mutuellement dans la clause **uses** de leur section interface. Ainsi, l'exemple suivant produit une erreur de compilation :

```
unit Unit1;
interface
uses Unit2;
⋮
unit Unit2;
interface
uses Unit1;
⋮
```

Par contre, les deux unités peuvent sans problème se référencer mutuellement si l'une des références est déplacée dans la section implémentation :

```
unit Unit1;
interface
uses Unit2;
⋮
unit Unit2;
interface
⋮
```

```
implementation
uses Unit1;
⋮
```

Afin de réduire les risques de références circulaires, il est préférable à chaque fois que cela est possible de lister les unités dans la clause **uses** de l'implémentation. C'est uniquement si des identificateurs d'une autre unité sont utilisés dans la section interface qu'il est nécessaire d'indiquer cette unité dans la clause **uses** de l'interface.

Chapitre

4

Eléments syntaxiques

Le Pascal Objet utilise une partie du jeu de caractères ASCII : les lettres de *A* à *Z* et de *a* à *z*, les chiffres de *0* à *9* et d'autres caractères standard. Il ne tient *pas* compte des différences majuscules/minuscules. Le caractère espace (ASCII 32) et les caractères de contrôle (ASCII 0 à 31, y compris le caractère ASCII 13 de retour ou de fin de ligne) sont appelés des *blancs*.

Les éléments syntaxiques fondamentaux, appelés des *tokens*, se combinent pour former des expressions, des déclarations et des instructions. Une *instruction* décrit une action algorithmique qui peut être exécutée dans un programme. Une *expression* est une unité syntaxique située dans une instruction et qui désigne une valeur. Une *déclaration* définit un identificateur (par exemple le nom d'une fonction ou d'une variable) qui peut être employé dans des expressions ou des instructions et, si c'est nécessaire, alloue de la mémoire à l'identificateur.

Eléments syntaxiques fondamentaux

Au niveau le plus élémentaire, un programme est une suite de tokens délimités par des séparateurs. Un *token* est la plus petite unité de texte significative d'un programme. Un *séparateur* est soit un blanc, soit un commentaire. Strictement parlant, il n'est pas toujours nécessaire de placer un séparateur entre deux tokens ; par exemple le fragment de code suivant :

```
Taille:=20;Prix:=10;
```

est parfaitement légal. Néanmoins, l'habitude et un souci de lisibilité suggèrent de l'écrire de la manière suivante :

```
Taille := 20;
Prix := 10;
```

Les tokens sont divisés en *symboles spéciaux*, *identificateurs*, *mots réservés*, *directives*, *nombres*, *labels* et *chaînes de caractères*. La chaîne de caractères est le seul token qui

peut contenir des séparateurs. Il doit obligatoirement y avoir un séparateur entre des identificateurs, mots réservés, nombres et labels adjacents.

Symboles spéciaux

Les symboles spéciaux sont des caractères, ou des paires de caractères, non alphanumériques ayant un sens déterminé. Les caractères individuels suivants sont des symboles spéciaux :

 # $ & ' ( ) * + , – . / : ; < = > @ [ ] ^ { }

Les paires de caractères suivantes sont également des symboles spéciaux :

(* (. *) .) .. // := <= >= <>

Le crochet ouvrant **[** est équivalent à la paire de caractères parenthèse ouvrante suivie d'un point **(.** ; le crochet fermant est équivalent à la paire de caractères point suivi d'une parenthèse fermante **.)**. Les paires parenthèse ouvrante-astérisque et astérisque-parenthèse fermante sont équivalentes aux accolades ouvrante et fermante { }.

Les caractères **!**, **"** (guillemet), **%**, **?**, \, _ (souligné), **|** et **~** (tilde) ne sont pas des caractères spéciaux.

Identificateurs

Les identificateurs désignent les constantes, variables, champs, types, propriétés, procédures, fonctions, programmes, unités, bibliothèques et paquets. La longueur d'un identificateur peut être quelconque, mais seuls les 255 premiers caractères sont significatifs. Un identificateur doit commencer par une lettre ou un souligné (_) et ne peut contenir d'espaces ; après le premier caractère, les chiffres, les lettres et le caractère souligné sont autorisés. Les mots réservés ne peuvent être utilisés comme identificateurs.

Comme le Pascal Objet ne tient pas compte des différences majuscules/minuscules, un identificateur comme *CalculateValue* peut être écrit dans n'importe laquelle des formes suivantes :

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

Dans Linux, les seuls identificateurs pour lesquels la distinction majuscules/minuscules est importante sont les noms d'unités. Comme les noms d'unités correspondent à des noms de fichiers, une incohérence de majuscules/minuscules peut affecter la compilation.

Identificateurs qualifiés

Quand vous utilisez un identificateur qui a été déclaré à plusieurs endroits, il est parfois nécessaire de *qualifier* l'identificateur. La syntaxe d'un identificateur qualifié est :

$identificateur_1.identificateur_2$

où $identificateur_1$ qualifie $identificateur_2$. Si, par exemple, deux unités déclarent une variable appelée *ValeurEnCours*, vous pouvez désigner *ValeurEnCours* de *Unit2* en écrivant :

```
Unit2.ValeurEncours
```

Il est possible de chaîner les qualificateurs. Par exemple :

```
Form1.Button1.Click
```

Appelle la méthode *Click* de *Button1* dans *Form1*.

Si vous ne qualifiez pas un identificateur, son interprétation est déterminée par les règles de portée décrites dans “Blocs et portée” à la page 4-30.

Mots réservés

Les mots réservés suivants ne peuvent être redéfinis ou utilisés comme identificateur :

Tableau 4.1 Mots réservés

and	**downto**	**in**	**or**	**string**
array	**else**	**inherited**	**out**	**then**
as	**end**	**initialization**	**packed**	**threadvar**
asm	**except**	**inline**	**procedure**	**to**
begin	**exports**	**interface**	**program**	**try**
case	**file**	**is**	**property**	**type**
class	**finalization**	**label**	**raise**	**unit**
const	**finally**	**library**	**record**	**until**
constructor	**for**	**mod**	**repeat**	**uses**
destructor	**function**	**nil**	**resourcestring**	**var**
dispinterface	**goto**	**not**	**set**	**while**
div	**if**	**object**	**shl**	**with**
do	**implementation**	**of**	**shr**	**xor**

Outre les mots spécifiés dans le tableau 4.1, **private**, **protected**, **public**, **published** et **automated** se comportent comme des mots réservés à l'intérieur d'une déclaration de type objet et sont traités ailleurs comme des directives. Les mots **at** et **on** ont aussi des significations spéciales.

Directives

Les directives sont des mots sensibles à certains endroits du code source. Elles ont une signification particulière en Pascal Objet, mais, à la différence des mots réservés elles n'apparaissent que dans des contextes où ne peuvent se placer des identificateurs définis par l'utilisateur. De ce fait, vous pouvez (même si cela est déconseillé) définir un identificateur identique à une directive.

Tableau 4.2 Directives

absolute	**dynamic**	**message**	**private**	**resident**
abstract	**export**	**name**	**protected**	**safecall**
assembler	**external**	**near**	**public**	**stdcall**
automated	**far**	**nodefault**	**published**	**stored**
cdecl	**forward**	**overload**	**read**	**varargs**
contains	**implements**	**override**	**readonly**	**virtual**
default	**index**	**package**	**register**	**write**
deprecated	**library**	**pascal**	**reintroduce**	**writeonly**
dispid	**local**	**platform**	**requires**	

Nombres

Les constantes entières et réelles peuvent être représentées en notation décimale comme une séquence de chiffres sans virgule ni espaces et préfixés par les opérateurs **+** ou **–** afin d'indiquer le signe. Par défaut les valeurs sont positives (ainsi `67258` est identique à `+67258`) et doivent se trouver à l'intérieur de l'intervalle du plus grand type prédéfini entier ou réel.

Les nombres avec un séparateur décimal ou un exposant désignent des réels, alors que les autres nombres désignent des entiers. Quand le caractère *E* ou *e* apparaît dans un réel, cela signifie "multiplié par dix à la puissance". Par exemple, `7E-2` signifie 7×10^{-2} de même `12.25e+6` et `12.25e6` signifient tous les deux 12.25×10^{6}.

L'utilisation d'un préfixe dollar, par exemple `$8F`, désigne une valeur hexadécimale. Le signe d'un hexadécimal est déterminé par le bit le plus à gauche (le moins significatif) de sa représentation binaire.

Pour davantage d'informations sur les types réel et entier, voir chapitre 5, "Types de données, variables et constantes". Pour davantage d'informations sur les types de données numériques, voir "Vraies constantes" à la page 5-44.

Labels

Un label est une suite d'au maximum quatre chiffres, c'est donc un nombre compris entre 0 et 9999. Les zéros à gauche ne sont pas significatifs. Les identificateurs peuvent également être utilisés comme labels.

Les labels sont utilisés dans les instructions **goto**. Pour davantage d'informations sur l'instruction **goto** et les labels, voir "Instructions goto" à la page 4-21.

Chaînes de caractères

Une chaîne de caractères (appelée également *littéral chaîne* ou *constante chaîne*) est une *chaîne délimitée*, une *chaîne de contrôle* ou une combinaison de chaînes délimitées et de chaînes de contrôle. Il ne peut y avoir de séparateur qu'à l'intérieur des chaînes délimitées.

Une chaîne délimitée est une séquence comportant jusqu'à 255 caractères du jeu de caractères ASCII étendu, écrits sur une seule ligne et placés entre apostrophes. Une chaîne délimitée ne contenant rien entre les apostrophes est une *chaîne vide*. Deux apostrophes consécutives à l'intérieur d'une chaîne délimitée désigne un seul caractère : une apostrophe. Par exemple :

```
'BORLAND'            { BORLAND }
'Vous l''aurez'      { Vous l'aurez }
''''                 { ' }
''                   { chaîne nulle }
' '                  { un espace }
```

Une chaîne de contrôle est une séquence constituée d'un ou de plusieurs *caractères de contrôle*. Un caractère de contrôle est constitué du symbole **#** suivi d'une constante entière non signée comprise entre 0 et 255 (en décimal ou en hexadécimal) ; il désigne le caractère ASCII correspondant. Ainsi, la chaîne de contrôle suivante :

```
#89#111#117
```

Est équivalent à la chaîne délimitée :

```
'You'
```

Il est possible de combiner les chaînes délimitées et les chaînes de contrôle pour constituer des chaînes plus longues. Par exemple, vous pouvez utiliser :

```
'Ligne 1'#13#10'Ligne 2'
```

pour placer une séquence retour chariot-passage à la ligne entre "Ligne 1" et "Ligne 2". Néanmoins, il n'est pas possible de concaténer deux chaînes délimitées en procédant ainsi, en effet deux apostrophes se suivant sont interprétées comme un seul caractère. Pour concaténer des chaînes délimitées, utilisez l'opérateur + décrit dans "Opérateurs de chaînes" à la page 4-10 ou combinez-les en une seule chaîne délimitée.

La *longueur* d'une chaîne de caractères est le nombre de caractères de la chaîne. Une chaîne de caractères de longueur quelconque est compatible avec tout type de chaîne et avec le type *PChar*. Une chaîne de caractères de longueur 1 est compatible avec tout type de caractère et, si la syntaxe étendue est activée (**{$X+}**), une chaîne de caractères non vide, de longueur $n \geq 1$ est compatible avec des tableaux basés sur zéro et des tableaux compactés de n caractères. Pour davantage d'informations sur les types de chaîne, voir chapitre 5, "Types de données, variables et constantes".

Commentaires et directives de compilation

Les commentaires ne sont pas pris en compte par le compilateur sauf quand ils agissent comme séparateur (en délimitant des tokens consécutifs) ou comme directive de compilation.

Il y a plusieurs manière de construire des commentaires :

```
{ Le texte placé entre l'accolades ouvrante et l'accolade fermante est un commentaire. }

(* Le texte placé entre une parenthèse ouvrante-astérisque et une astérisque-parenthèse
fermante est également un commentaire. *)

// Tout texte placé entre une double barre oblique et la fin de la ligne
// est un commentaire.
```

Un commentaire contenant le signe dollar (**$**) immédiatement après **{** ou **(*** est une directive de compilation. Par exemple ;

```
{$WARNINGS OFF}
```

indique au compilateur de ne pas générer de messages d'avertissement.

Expressions

Une *expression* est une construction qui renvoie une valeur. Par exemple :

```
X                   { variable }
@X                  { adresse d'une variable }
15                  { constante entière}
TauxInteret         { variable }
Calc(X,Y)           { appel de fonction}
X * Y               { produit de X par Y }
Z / (1 - Z)         { quotient de Z par (1 - Z) }
X = 1.5             { booléen }
C in Range1         { booléen }
not Fini            { négation d'un booléen }
['a','b','c']       { ensemble }
Char(48)            { transtypage de valeur }
```

Les expressions les plus simples sont les variables et les constantes ; pour une description plus détaillée, voir chapitre 5, “Types de données, variables et constantes”). Les expressions complexes sont construites à partir d'expressions plus simples utilisant des *opérateurs*, des *appels de fonction*, *des constructeurs d'ensemble*, *des indices* et des *transtypages*.

Opérateurs

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

opérateurs sont @, **not**, **^**, *****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**, **+**, **–**, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **in** et **is**.

Les opérateurs @, **not** et **^** sont des opérateurs *unaire* (n’utilisant qu’un seul opérande). Tous les autres opérateurs sont *binaires* (ils utilisent deux opérandes), à l’exception de **+** et **–** qui peuvent être unaire ou binaire. Un opérateur unaire précède toujours son opérande (par exemple `-B`) à l’exception de **^** qui suit son opérande (par exemple, `P^`). Un opérateur binaire est placé entre ses opérandes (par exemple, `A = 7`).

Certains opérateurs se comportent différemment selon le type des données transmises. Ainsi, **not** effectue une négation bit-à-bit pour un opérande entier et une négation logique pour un opérande booléen. De tels opérateurs apparaissent dans plusieurs des catégories indiquées plus bas.

A l’exception de **^**, **is** et de **in**, tous les opérateurs acceptent des opérandes de type *Variant*. Pour davantage d’informations, voir “Types variants” à la page 5-33.

Les sections suivantes supposent une certaine connaissance des types de données Pascal Objet. Pour davantage d’informations sur les types de données, voir chapitre 5, “Types de données, variables et constantes”.

Pour davantage d’informations sur la priorité des opérateurs dans les expressions complexes, voir “Règles de priorité des opérateurs” à la page 4-14.

Opérateurs arithmétiques

Les opérateurs arithmétiques suivants attendent des opérandes réels ou entiers : **+**, **–**, *****, **/**, **div** et **mod**.

Tableau 4.3 Opérateurs arithmétiques binaires

Opérateur	Opération	Types d’opérande	Type du résultat	Exemple
+	addition	entier, réel	entier, réel	`X + Y`
–	soustraction	entier, réel	entier, réel	`Result - 1`
*****	multiplication	entier, réel	entier, réel	`P * InterestRate`
/	division réelle	entier, réel	réel	`X / 2`
div	division entière	entier	entier	Total **div** UnitSize
mod	reste	entier	entier	Y **mod** 6

Tableau 4.4 Opérateurs arithmétiques unaires

Opérateur	Opération	Types d’opérande	Type du résultat	Exemple
+	signe identité	entier, réel	entier, réel	`+7`
–	signe négation	entier, réel	entier, réel	`-X`

Les règles suivantes s'appliquent aux opérateurs arithmétiques :

- La valeur de *x*/*y* est de type *Extended*, indépendamment du type de *x* et *y*. Pour les autres opérateurs, le résultat est de type *Extended* dès qu'au moins un des opérandes est de type réel ; sinon le résultat est de type *Int64* quand au moins un des opérandes est de type *Int64* ; sinon le résultat est de type *Integer*. Si le type d'un opérande est un sous-intervalle d'un type entier, il est traité comme étant de ce type entier.
- La valeur de *x* **div** *y* est la valeur de *x*/*y* arrondi vers le bas à l'entier le plus proche.
- L'opérateur **mod** renvoie le reste obtenu par la division de ses opérandes. En d'autres termes : *x* **mod** *y* = *x* – (*x* **div** *y*) * *y*.
- Il y a une erreur d'exécution si *y* vaut zéro dans une expression de la forme *x*/*y*, *x* **div** *y* ou *x* **mod** *y*.

Opérateurs booléens

Les opérateurs booléens **not**, **and**, **or** et **xor** prennent des opérandes de tout type booléen et renvoie une valeur de type *Boolean*.

Tableau 4.5 Opérateurs booléen

Opérateur	Opération	Types d'opérande	Types du résultat	Exemple
not	négation	booléen	*Boolean*	**not** (C **in** MySet)
and	conjonction	booléen	*Boolean*	Done **and** (Total > 0)
or	disjonction	booléen	*Boolean*	A **or** B
xor	disjonction exclusive	booléen	*Boolean*	A **xor** B

Ces opérations sont régies par les règles standard de la logique booléenne. Par exemple, une expression de la forme *x* **and** *y* vaut *True* si et seulement si *x* et *y* valent tous les deux *True*.

Evaluation booléenne optimisée ou complète

Le compilateur gère deux modes d'évaluation des opérateurs **and** et **or** : l'évaluation complète et l'évaluation optimisée (partielle). *Evaluation complète* signifie que chaque opérande d'une conjonction ou d'une disjonction est évaluée même si le résultat de l'expression entière est déjà déterminé. *Evaluation optimisée* signifie que l'évaluation se fait strictement de gauche à droite et s'arrête dès que le résultat de l'expression entière est connu. Ainsi, quand l'expression `A and B` est évaluée en mode optimisé alors que *A* vaut *False*, le compilateur n'évalue même pas *B* ; il sait que la valeur de l'expression est *False* dès que *A est évalué*.

L'évaluation optimisée est généralement préférable car elle garantit une durée d'exécution minimum et le plus souvent une taille de code minimum. L'évaluation complète est parfois commode quand un opérande est une fonction ayant des effets secondaires jouant un rôle dans l'exécution du programme.

L'évaluation optimisée permet également l'utilisation de constructions qui pourraient provoquer des erreurs à l'exécution. Par exemple, le code suivant parcourt la chaîne *S*, jusqu'à la première virgule :

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
  ⋮
  Inc(I);
end;
```

Si jamais *S* ne contient pas de virgule, la dernière itération incrémente *I* à une valeur supérieure à la longueur de *S*. Lors du dernier test de la condition du **while**, une évaluation complète entraîne une tentative de lecture de *S[I]*, ce qui peut provoquer une erreur d'exécution. Avec une évaluation optimisée, la deuxième partie de la condition du **while** `(S[I] <> ',')` n'est pas évaluée quand la première partie est fausse.

Utilisez la directive de compilation **$B** pour contrôler le mode d'évaluation. L'état par défaut est **{$B–}**, qui active l'évaluation optimisée. Pour activer localement l'évaluation complète, ajoutez la directive **{$B+}** dans votre code. Vous pouvez également passer en évaluation complète pour un projet en sélectionnant Evaluation booléenne comp dans la page Compilateur de la boîte de dialogue Options de projet.

Remarque Si un opérande quelconque fait appel à un variant, le compilateur effectue toujours une évaluation complète (même dans l'état **{$B–}**).

Opérateurs logiques bit-à-bit

Les opérateurs logiques suivants effectuent des manipulations bit-à-bit sur des opérandes entiers. Si, par exemple, la valeur stockée dans *X* est (au format binaire) 001101 et si la valeur stockée dans *Y* est 100001, l'instruction

```
Z := X or Y;
```

affecte la valeur 101101 à *Z*.

Tableau 4.6 Opérateurs logiques bit-à-bit

Opérateur	Opération	Types d'opérande	Type du résultat	Exemples
not	négation bit-à-bit	entier	entier	**not** `X`
and	et bit-à-bit	entier	entier	`X` **and** `Y`
or	ou bit-à-bit	entier	entier	`X` **or** `Y`
xor	ou exclusif bit-à-bit	entier	entier	`X` **xor** `Y`
shl	décalage à gauche bit-à-bit	entier	entier	`X` **shl** `2`
shr	décalage à droite bit-à-bit	entier	entier	`Y` **shl** `I`

Les règles suivantes s'appliquent aux opérateurs bit-à-bit :

- Le résultat d'une opération **not** est de même type que l'opérande.
- Si les opérandes d'une opération **and**, **or ou xor** sont tous les deux des entiers, le résultat est du plus petit type entier prédéfini qui inclut toutes les valeurs possibles des deux types.
- Les opérations *x* **shl** *y* et *x* **shr** *y* décalent la valeur *x* vers la gauche ou vers la droite de *y* bits, ce qui revient à multiplier ou à diviser *x* par 2^y ; le résultat est du même type que *x*. Si, par exemple, *N* contient la valeur 01101 (13 en décimal), alors `N shl 1` renvoie 11010 (26 en décimal).

Opérateurs de chaînes

Les opérateurs relationnels **=**, **<>**, **<**, **>**, **<=** et **>=** acceptent tous des opérandes chaîne (voir "Opérateurs relationnels" à la page 4-12). L'opérateur **+** concatène deux chaînes.

Tableau 4.7 Opérateurs de chaînes

Opérateur	Opération	Types d'opérande	Type du résultat	Exemple
+	concaténation	chaîne, chaîne compactée, caractère	chaîne	`S + '. '`

Les règles suivantes s'appliquent à la concaténation de chaîne :

- Les opérandes pour l'opérateur **+** peuvent être des chaînes, des chaînes compactées (des tableaux compactés de type *Char*) ou des caractères. Cependant, si un opérande est de type *WideChar*, l'autre opérande doit être une chaîne longue.
- Le résultat d'une opération **+** est compatible avec tout type de chaîne. Cependant, si les opérandes sont tous deux des chaînes courtes ou des caractères, et si leur longueur cumulée est supérieure à 255, le résultat est tronqué aux 255 premiers caractères.

Opérateurs de pointeurs

Les opérateurs relationnels **<**, **>**, **<=** et **>=** acceptent tous des opérandes de type *PChar* Les opérateurs relationnels **=**, **<>**, **<**, **>**, **<=** et **>=** acceptent tous des opérandes chaîne (voir "Opérateurs relationnels" à la page 4-12). Les opérateurs suivants acceptent également des pointeurs comme opérandes. Pour davantage d'informations sur les pointeurs, voir "Pointeurs et types pointeur" à la page 5-27.

Tableau 4.8 Opérateurs de pointeurs

Opérateur	Opération	Types d'opérande	Type du résultat	Exemple
+	addition de pointeurs	pointeur de caractère, entier	pointeur de caractère	`P + I`
-	soustraction de pointeurs	pointeur de caractère, entier	pointeur de caractère, entier	`P - Q`

Tableau 4.8 Opérateurs de pointeurs

Opérateur	Opération	Types d'opérande	Type du résultat	Exemple
^	déréférencement de pointeur	pointeur	type de base du pointeur	`P^`
=	égalité	pointeur	*Boolean*	`P = Q`
<>	inégalité	pointeur	*Boolean*	`P <> Q`

L'opérateur **^** déréférence un pointeur. Son opérande peut être un pointeur de type quelconque sauf un pointeur générique (*Pointer)* qui doit être transtypé avant d'être déréférencé.

P **=** Q vaut *True* si P et Q pointent sur la même adresse ; sinon, P **<>** Q est *True*.

Vous pouvez utiliser les opérateurs **+** et **–** pour incrémenter ou décrémenter le décalage d'un pointeur de caractère. Vous pouvez également utiliser **–** pour calculer la différence de décalage entre deux pointeurs de caractère. Les règles suivantes s'appliquent :

- Si I est un entier et P un pointeur de caractère, alors $P + I$ ajoute I à l'adresse spécifiée par P ; c'est-à-dire que le résultat est un pointeur sur l'adresse I caractères après P. L'expression $I + P$ est équivalente à $P + I$. $P - I$ soustrait I de l'adresse spécifiée par P ; c'est-à-dire que le résultat est un pointeur sur l'adresse I caractères avant P.
- Si P et Q sont tous les deux des pointeurs de caractère, $P - Q$ calcule la différence entre l'adresse spécifiée par P (l'adresse la plus haute) et l'adresse spécifiée par Q (l'adresse la plus basse) ; c'est-à-dire que le résultat est un entier qui indique le nombre de caractères séparant P de Q. *L'opération* $P + Q$ n'est pas définie.

Opérateurs d'ensembles

Les opérateurs suivants acceptent des ensembles comme opérandes :

Tableau 4.9 Opérateurs d'ensembles

Opérateur	Opération	Types d'opérande	Type du résultat	Exemple
+	union	ensemble	ensemble	`Set1 + Set2`
–	différence	ensemble	ensemble	`S - T`
*	intersection	ensemble	ensemble	`S * T`
<=	sous-ensemble	ensemble	*Boolean*	`Q <= MySet`
>=	sur-ensemble	ensemble	*Boolean*	`S1 >= S2`
=	égalité	ensemble	Boolean	`S2 = MySet`
<>	différence	ensemble	Boolean	`MySet <> S1`
in	inclusion	scalaire, ensemble	*Boolean*	`A` **in** `Set1`

Les règles suivantes s'appliquent aux opérateurs **+**, **–** et *****:

- Le scalaire *O* appartient à *X* **+** *Y* si et seulement si *O* est dans *X* ou dans *Y* (ou dans les deux). *O* appartient à *X* **–** *Y* si et seulement si *O* est dans *X* mais pas dans *Y*. *O* appartient à *X* ***** *Y* si et seulement si *O* est dans *X* et dans *Y*.
- Le résultat d'une opération **+**, **–** ou ***** est de type **set of** *A*..*B*, où *A* est la plus petite valeur scalaire de l'ensemble résultant et *B* la plus grande.

Les règles suivantes s'appliquent aux opérateurs **<=**, **>=**, **=**, **<>** et **in** :

- *X* **<=** *Y* vaut *True* si chaque membre de *X* est également membre de *Y* ; *Z* **>=** *W* est équivalent à *W* **<=** *Z*. *U* **=** *V* vaut *True* si *U* et *V* contiennent exactement les mêmes membres ; sinon, *U* **<>** *V* vaut *True*.
- Pour un scalaire *O* et un ensemble *S*, *O* **in** *S* vaut *True* uniquement si *O* est membre de *S*.

Opérateurs relationnels

Les opérateurs relationnels sont utilisés pour comparer deux opérandes. Les opérateurs **=**, **<>**, **<=** et **>=** s'appliquent également aux ensembles (voir "Opérateurs d'ensembles" à la page 4-11) ; **=** et **<>** s'appliquent également aux pointeurs (voir "Opérateurs de pointeurs" à la page 4-10).

Tableau 4.10 Opérateurs relationnels

Opérateur	Opération	Types d'opérande	Type du résultat	Exemple
=	égalité	simple, classe, référence de classe, interface, chaîne, chaîne compactée	*Boolean*	`I = Max`
<>	différence	simple, classe, référence de classe, interface, chaîne, chaîne compactée	*Boolean*	`X <> Y`
<	inférieur à	simple, chaîne, chaîne compactée, *PChar*	*Boolean*	`X < Y`
>	supérieur à	simple, chaîne, chaîne compactée, *PChar*	*Boolean*	`Len > 0`
<=	inférieur ou égal à	simple, chaîne, chaîne compactée, *PChar*	*Boolean*	`Cnt <= I`
>=	supérieur ou égal à	simple, chaîne, chaîne compactée, *PChar*	*Boolean*	`I >= 1`

Dans la plupart des cas simples, la comparaison est évidente. Par exemple *I* **=** *J* vaut *True* uniquement si *I* et *J* ont la même valeur, sinon *I* **<>** *J* vaut *True*. Les règles suivantes s'appliquent aux opérateurs de comparaison :

- Les opérandes doivent être de types compatibles, sauf pour un réel et un entier qui peuvent être comparés.
- Les chaînes sont comparées en fonction de l'ordre du jeu de caractères ASCII étendu. Les types caractère sont traités comme des chaînes de longueur 1.
- Deux chaînes compactées doivent avoir le même nombre de composants pour pouvoir être comparées. Quand une chaîne compactée ayant *n* composants est

comparée à une chaîne, la chaîne compactée est traitée comme une chaîne de longueur *n*.

- Les opérateurs <, >, <= et >= s'appliquent à des opérandes *PChar* uniquement si les deux pointeurs désignent le même tableau de caractères.
- Les opérateurs = et <> acceptent des opérandes de type classe ou référence de classe. Avec des opérandes de type classe, = et <> sont évalués en utilisant les règles s'appliquant aux pointeurs : *C* = *D* vaut *True* uniquement si *C* et *D* pointent sur la même instance d'objet ; sinon *C* <> *D* vaut *True*. Avec des opérandes de type référence de classe, *C* = *D vaut True* uniquement si *C* et *D* désignent la même classe, sinon *C* <> *D* vaut *True*. Pour davantage d'informations sur les classes, voir chapitre 7, "Classes et objets".

Opérateurs de classes

Les opérateurs **as** et **is** prennent des classes et des instances d'objet comme opérandes ; **as** accepte également des interfaces. Pour davantage d'informations, voir chapitre 7, "Classes et objets" et chapitre 10, "Interfaces d'objets".

Les opérateurs relationnels = et <> opèrent également sur des classes. Voir "Opérateurs relationnels" à la page 4-12.

L'opérateur @

L'opérateur @ renvoie l'adresse d'une variable, d'une fonction, d'une procédure ou d'une méthode ; @ construit un pointeur sur son opérande. Pour davantage d'informations sur les pointeurs, voir "Pointeurs et types pointeur" à la page 5-27.

Les règles suivantes s'appliquent à l'opérateur @ :

- Si *X* est une variable, @*X* renvoie l'adresse de *X*. Des règles spéciales s'appliquent quand *X* est une variable de type procédure ; voir "Types procédure dans les instructions et les expressions" à la page 5-32.) @*X* est de type *Pointer* si la directive de compilation par défaut **{$T–}** est active. Dans l'état **{$T+}** @*X* est de type **^***T*, où *T* est le type de *X*.
- Si *F est une* routine (une fonction ou une procédure), @*F* renvoie le point d'entrée de *F*. @*F* est toujours *de type Pointer*.
- Quand @ est appliqué à une méthode définie dans une classe, l'identificateur de la méthode doit être qualifié par le nom de la classe. Par exemple :

  ```
  @TMyClass.DoSomething
  ```

  pointe sur la méthode *DoSomething* de *TMyClass*. Pour davantage d'informations sur les classes et les méthodes, voir chapitre 7, "Classes et objets".

Règles de priorité des opérateurs

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

Tableau 4.11 Priorité des opérateurs

Opérateurs	Priorité
@, **not**	première (maximum)
*, /, **div**, **mod**, **and**, **shl**, **shr**, **as**	seconde
+, –, **or**, **xor**	troisième
=, <>, <, >, <=, >=, **in**, **is**	quatrième (minimum)

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

```
X + Y * Z
```

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

```
X - Y + Z
```

commence par soustraire *Y* à *X* puis ajoute *Z* au résultat : **–** et **+** ayant la même priorité, l'opération de gauche est effectuée en premier.

Vous pouvez utiliser des parenthèses pour redéfinir ces règles de priorité. Une expression entre parenthèses est tout d'abord évaluée puis traitée comme un seul opérande. Par exemple :

```
(X + Y) * Z
```

multiplie *Z* par la somme de *X* et *Y*.

Les parenthèses sont parfois nécessaires dans des situations où, au premier regard elles ne sembles pas utiles. Par exemple, soit l'expression :

```
X = Y or X = Z
```

L'interprétation voulue est manifestement :

```
(X = Y) or (X = Z)
```

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

```
(X = (Y or X)) = Z
```

Ce qui provoque une erreur de compilation sauf si *Z* est un booléen.

Les parenthèses rendent souvent le code plus simple à écrire et à lire même quand elles sont, techniquement parlant, inutiles. Ainsi le premier exemple peut également s'écrire :

```
X + (Y * Z)
```

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

Appels de fonctions

Comme les fonctions renvoient une valeur, les appels de fonction sont des expressions. Si, par exemple, vous avez défini une fonction appelée *Calc* qui attend deux arguments entiers et qui renvoie un entier, l'appel de fonction `Calc(24, 47)` est une expression entière. Si *I* et *J* sont des variables entières, alors `I + Calc(J, 8)` est également une expression entière. Voici quelques exemples d'appels de fonction :

```
Somme(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Rayon, Hauteur)
ExtraitValeur
TUnObjet.UneMethode(I,J);
```

Pour davantage d'informations sur les fonctions, voir chapitre 6, “Procédures et fonctions”.

Constructeurs d'ensembles

Un constructeur d'ensemble désigne une valeur de type ensemble. Par exemple :

```
[5, 6, 7, 8]
```

désigne l'ensemble dont les membres sont 5, 6, 7 et 8. Le constructeur d'ensemble :

```
[ 5..8 ]
```

désigne le même ensemble.

La syntaxe d'un constructeur d'ensemble est :

[$élément_1$, ..., $élément_n$]

où chaque *élément* est soit une expression désignant un scalaire ayant le type de base de l'ensemble ou une paire de telles expressions avec deux points (**..**) entre. Quand un *élément* est de la forme *x*..*y*, c'est un abrégé pour tous les scalaires compris dans l'intervalle allant de *x* à *y*, bornes incluses ; mais si *x* est supérieur à *y*, alors *x*..*y* ne désigne rien et **[***x*..*y***]** est l'ensemble vide. Le constructeur d'ensemble `[ ]` désigne l'ensemble vide alors que **[***x***]** désigne l'ensemble dont le seul membre est la valeur de *x*.

Voici des exemples de constructeurs d'ensembles :

```
[rouge, vert, MaCouleur]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

Pour davantage d'informations sur les ensembles, voir "Ensembles" à la page 5-18.

Indices

Les chaînes de caractères, les tableaux et propriétés tableau et les pointeurs sur des chaînes ou des tableaux peuvent être indicés. Si, par exemple, *NomFichier* est une variable chaîne, l'expression `NomFichier[3]` renvoie le troisième caractère de la chaîne désignée par *NomFichier*, alors que `NomFichier[I + 1]` renvoie le caractère suivant immédiatement celui indicé par *I*. Pour des informations sur les chaînes, voir "Types chaîne" à la page 5-11. Pour des informations sur les tableaux et les propriétés tableau, voir "Tableaux" à la page 5-19 et "Propriétés tableau" à la page 7-21.

Transtypage

Il est parfois utile de traiter une expression ayant un type donné comme si elle était d'un type différent. Le transtypage permet de le faire en modifiant, temporairement, le type d'une expression. Par exemple, `Integer('A')` transtype le caractère *A* en un entier.

La syntaxe du transtypage est :

IdentificateurType(*expression*)

Si l'expression est une variable, le résultat est appelé un *transtypage de variable* ; sinon le résultat est un *transtypage de valeur*. Si la syntaxe est la même, des règles différentes s'appliquent à ces deux sortes de transtypages.

Transtypage de valeur

Dans un transtypage de valeur, l'identificateur de type et l'expression à transtyper doivent être tous les deux de type scalaire ou tous les deux de type pointeur. Voici des exemples de transtypage de valeur :

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Tampon)
```

La valeur résultante est obtenue en convertissant l'expression entre parenthèses. Cela peut nécessiter une réduction ou une extension si la taille du type spécifié diffère de celle de l'expression. Le signe de l'expression est toujours conservé.

L'instruction :

```
I := Integer('A');
```

affecte la valeur de `Integer('A')`—c'est-à-dire 65— à la variable *I*.

Un transtypage de valeur ne peut être suivi par un qualificateur et ne peut apparaître dans la partie gauche d'une instruction d'affectation.

Transtypage de variable

Il est possible de convertir toute variable vers un type quelconque, à partir du moment où la taille est la même et que vous ne mélangez pas des entiers et des réels. Pour convertir des types numériques, utilisez les fonctions standard comme *Int* ou *Trunc*. Voici des exemples de transtypage de variable :

```
Char(I)
Boolean(Compteur)
TUnTypeDefini(MaVariable)
```

Le transtypage de variable peut apparaître des deux côtés d'une affectation. Par exemple :

```
var MonCar: char;
⋮
Shortint(MonCar) := 122;
```

affecte le caractère *z* (ASCII 122) à *MonCar*.

Il est possible de convertir des variables dans un type procédure. Par exemple, étant donné la déclaration :

```
type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

Les affectations suivantes sont possibles :

```
F := Func(P);          { Affecte la valeur procédurale de P à F }
Func(P) := F;          { Affecte la valeur procédurale de F à P }
@F := P;               { Affecte la valeur de pointeur de P à F }
P := @F;               { Affecte la valeur de pointeur de F à P }
N := F(N);             { Appel de la fonction via F }
N := Func(P)(N);       { Appel de la fonction via P }
```

Il est possible de faire suivre un transtypage de variable par un qualificateur, comme dans l'exemple suivant :

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;
  TWordRec = record
    Low, High: Word;
  end;
  PByte = ^Byte;
var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;

begin
  W := $1234;
  B := TByteRec(W).Lo;
```

```
  TByteRec(W).Hi := 0;
  L := $01234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

Dans cet exemple, *TByteRec* est utilisé pour accéder aux octets de poids faible et de poids fort d'un mot et *TWordRec* pour accéder aux mots de poids faible et de poids fort d'un entier long. Vous pouvez utiliser les fonctions prédéfinies *Lo* et *Hi* pour obtenir le même résultat, mais un transtypage de variable présente l'avantage de pouvoir s'utiliser dans la partie gauche d'une affectation.

Pour des informations sur le transtypage de pointeurs, voir “Pointeurs et types pointeur” à la page 5-27. Pour des informations sur le transtypage de classe et les types d'interface, voir “Opérateur as” à la page 7-27 et “Transtypage d'interfaces” à la page 10.

Déclarations et instructions

A part la clause **uses** (et des mots réservés qui comme **implementation** délimitent les sections d'une unité), un programme est entièrement constitué de *déclarations* et *d'instructions* qui sont organisées en *blocs*.

Déclarations

Les noms de variables, constantes, types, champs, propriétés, procédures, fonctions, programmes, unités, bibliothèques et paquets sont appelés des *identificateurs*. Les constantes numériques comme `26057` ne sont pas des identificateurs. Les identificateurs doivent être *déclarés* avant de pouvoir être utilisés. Seules exceptions à cette règle : quelques types sont prédéfinis, certaines routines et constantes sont reconnues automatiquement par le compilateur, la variable *Result* quand elle est utilisée dans un bloc de fonction et la variable *Self* quand elle est utilisée à l'intérieur d'une implémentation de méthode.

Une déclaration définit un identificateur et, si c'est approprié, lui alloue de la mémoire. Par exemple :

```
var Taille: Extended;
```

déclare une variable appelée *Taille* contenant une valeur *Extended* (réelle). De même :

```
function Agir(X, Y: string): Integer;
```

déclare une fonction appelée *Agir* qui attend deux chaînes en argument et renvoie un entier. Chaque déclaration se termine par un point-virgule. Quand vous déclarez en même temps plusieurs variables, constantes, types ou labels, il suffit de n'écrire qu'une seule fois le mot réservé approprié :

```
var
  Taille: Extended;
```

```
Quant: Integer;
Description: string;
```

La syntaxe et la position d'une déclaration dépendent du type d'identificateur défini. En général, les déclarations ne peuvent être faites qu'au début d'un bloc ou au début des sections interface ou implémentation d'une unité (après la clause **uses**). Les conventions spécifiques à la déclaration de variables, constantes, types, fonctions, etc., sont présentées dans les chapitres abordant ces sujets.

Les directives "de conseil" **platform**, **deprecated** et **library** peuvent être ajoutées à n'importe quelle déclaration, sauf que ces unités ne peuvent pas être déclarées avec **deprecated**. Dans le cas de la déclaration d'une procédure ou d'une fonction, la directive de conseil doit être séparée par un point virgule du reste de la déclaration. Exemples :

```
procedure SomeOldRoutine; stdcall; deprecated;

var VersionNumber: Real library;

type AppError = class(Exception)
  ⋮
end platform;
```

Quand le code source est compilé dans l'état **{$HINTS ON} {$WARNINGS ON}**, chaque référence à un identificateur déclaré avec l'une de ces directives génère un conseil ou un avertissement approprié. Utilisez **platform** pour spécifier des éléments propres à un environnement d'exploitation (Windows ou Linux), **deprecated** pour indiquer qu'un élément est obsolète, ou pris en charge uniquement pour des raisons de compatibilité, et **library** pour marquer les dépendances d'une bibliothèque ou d'une architecture de composants particulières (comme VCL ou CLX).

Instructions

Les instructions définissent des actions algorithmiques à l'intérieur d'un programme. Les instructions simples, comme les affectations ou les appels de procédure, peuvent se combiner pour former des boucles, des instructions conditionnelles ou d'autres instruction structurées.

Les instructions d'un bloc et des sections initialisation et finalisation d'une unité sont séparées par des points-virgules.

Instructions simples

Une instruction simple ne contient pas d'autre instruction. Les instructions simples sont les affectations, les appels de procédures et de fonctions et les déplacements **goto**.

Instructions d'affectation

Une instruction d'affectation a la forme :

variable := *expression*

où *variable* est une référence de variable quelconque (une variable, une variable transtypée, un pointeur déréférencé ou un composant d'une variable structurée) et *expression* est une expression compatible avec l'affectation. A l'intérieur d'un bloc de fonction, *variable* peut être remplacé par le nom de la fonction définie. Voir chapitre 6, "Procédures et fonctions".) Le symbole **:=** est parfois appelé *l'opérateur d'affectation*.

Une instruction d'affectation remplace la valeur en cours de *variable* par la valeur de *expression*. Par exemple ;

```
I := 3;
```

affecte la valeur 3 à la variable *I*. La référence de variable de la partie gauche de l'affectation peut apparaître dans l'expression à droite. Par exemple :

```
I := I + 1;
```

incrémente la valeur de *I*. Voici d'autres exemples d'instructions d'affectation :

```
X := Y + Z;
Fini := (I >= 1) and (I < 100);
Teinte := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MonCar) := 122;
TByteRec(W).Hi := 0;
MaChaine[I] := 'A';
UnTableau[I + 1] := P^;
TMonObjet.UnePropriete := True;
```

Appels de procédures et de fonctions

Un appel de procédure est constitué du nom de la procédure (avec ou sans qualificateur) suivi d'une liste de paramètres (si nécessaire). Voici quelques exemples

```
ImprimeDebut;
Transposer(A, N, M);
Trouver(Smith, William);
Writeln('Hello world!');
FaireUnTruc();
Unit1.UneProcedure;
TMonObjet.UneMethode(X,Y);
```

Les appels de fonction, comme les appels de procédure peuvent être traités comme des instructions :

```
MaFonction(X);
```

Quand vous utilisez un appel de fonction de cette manière, la valeur renvoyée est perdue.

Pour davantage d'informations sur les procédures et fonctions, voir chapitre 6, "Procédures et fonctions".

Instructions goto

Une instruction **goto**, qui a la forme :

```
goto label
```

transfère l'exécution du programme à l'instruction marquée par le label spécifié. Pour marquer une instruction, il faut tout d'abord déclarer le label. Il faut ensuite faire précéder l'instruction à marquer par le label et deux points :

label: *instruction*

Déclarez les labels de la manière suivante :

```
label label;
```

Vous pouvez déclarer plusieurs labels à la fois :

```
label label1, ..., labeln;
```

Un label peut être un identificateur légal quelconque ou un nombre compris entre 0 et 9999.

La déclaration de label, l'instruction marquée et l'instruction **goto** doivent se trouver dans le même bloc. Pour davantage d'informations sur les blocs, voir "Blocs et portée" à la page 4-30. Il n'est donc pas possible en utilisant **goto** de sortir d'une procédure ou d'une fonction. Ne marquez qu'une seule instruction d'un bloc avec le même label.

Par exemple :

```
label Debut;
⋮
Debut: Beep;
goto Debut;
```

crée une boucle infinie qui appelle la procédure *Beep*.

L'utilisation de l'instruction **goto** est généralement déconseillée en programmation structurée. Elle peut néanmoins être utilisée dans certains cas pour sortir de boucles imbriquées comme dans l'exemple suivant :

```
procedure TrouverPremiereReponse;
var X, Y, Z, Compteur: Integer;
label ReponseTrouvee;
begin
  Compteur := UneConstante;
  for X := 1 to Compteur do
    for Y := 1 to Compteur do
      for Z := 1 to Compteur do
        if ... { une condition portant sur X, Y et Z } then
          goto ReponseTrouvee;

  ⋮ {code à exécuter si aucune réponse n'est trouvée }
  Exit;

  ReponseTrouvee:
    ⋮ { code à exécuter si une réponse est trouvée }
end;
```

Remarquez que l'instruction **goto** sert à *sortir* de la boucle imbriquée. Ne rentrez jamais *dans* une boucle ou dans d'autres instructions structurées car cela peut donner des résultats imprévisibles.

Instructions structurées

Les instructions structurées sont composées à partir d'autres instructions. Utilisez une instruction structurée quand vous voulez exécuter certaines instructions de manière séquentielle, conditionnelle ou répétitive.

- Une instruction composée ou une instruction **with** exécute simplement la suite des instructions qui la compose.
- Une instruction conditionnelle (c'est-à-dire une instruction **if** ou **case**) exécute certains de ces composants, selon le critère spécifié.
- Une instruction de boucle (**repeat**, **while et** les boucles **for**) exécute de manière répétitive la suite des instructions qui la compose.
- Une instruction de groupe spéciale (**raise**, **les constructions try...except** et **try...finally**) crée et gère des *exceptions*. Pour des informations sur la génération et la gestion d'exceptions, voir "Exceptions" à la page 7-28.

Instructions composées

Une instruction composée est une suite d'instructions (simples ou structurées) qui doivent être exécutées dans l'ordre de leur écriture. L'instruction composée est délimitée par les mots réservés **begin** et **end** ; les instructions qui la composent sont séparées par des points-virgules. Par exemple :

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

Le dernier point-virgule avant le **end** est facultatif. Le code peut donc être écrit :

```
begin
  Z := X;
  X := Y;
  Y := Z
end;
```

Les instructions composées sont essentielles dans les contextes où la syntaxe Pascal Objet exige une seule instruction. En dehors des blocs de programmes, fonctions, ou procédures, elles peuvent se trouver dans d'autres instructions structurées, comme les instructions conditionnelles ou les boucles. Par exemple :

```
begin
  I := UneConstante;
  while I > 0 do
  begin
    ⋮
    I := I - 1;
```

```
  end;
end;
```

Il est possible d'écrire une instruction composée qui n'est constituée que d'une seule instruction ; comme les parenthèses dans une expression complexe, **begin** et **end** servent parfois à éviter des ambiguïtés et à améliorer la lisibilité du code. Vous pouvez également créer une instruction composée vide pour créer un bloc ne faisant rien :

```
begin
end;
```

Instructions With

Une instruction **with** est un raccourci permettant de référencer les champs d'un enregistrement ou les propriétés et méthodes d'un objet. L'instruction **with** a la syntaxe suivante :

with *obj* do *instruction*

ou

with obj_1, ..., obj_n do *instruction*

où *obj* est une référence de variable désignant un objet ou un enregistrement et *instruction* est une instruction simple ou structurée. A l'intérieur de *instruction*, vous pouvez faire référence aux champs, propriétés et méthodes de *obj* en utilisant seulement leur identificateur, sans utiliser de qualificateur.

Par exemple, étant donné les déclarations suivantes :

```
type TDate = record
  Jour: Integer;
  Mois: Integer;
  Annee: Integer;
end;

var DateCommande: TDate;
```

Vous pouvez écrire l'instruction **with** suivante :

```
with DateCommande do
  if Mois = 12 then
  begin
    Mois := 1;
    Annee := Annee + 1;
  end
  else
    Mois := Mois + 1;
```

Qui est équivalente à

```
if DateCommande.Mois = 12 then
begin
  DateCommande.Mois := 1;
  DateCommande.Annee := DateCommande.Annee + 1;
end
else
  DateCommande.Mois := DateCommande.Mois + 1;
```

Si l'interprétation de *obj* suppose des indices de tableau ou le déréférencement de pointeurs, ces actions ne sont effectuées qu'une seule fois, avant l'exécution de l'*instruction*. Cela rend les instructions **with** aussi efficaces que concises. Mais cela signifie également que les affectations d'une variable à l'intérieur de *l'instruction* ne peuvent changer l'interprétation de *obj* pendant l'exécution en cours de l'instruction **with**.

Chaque référence de variable ou nom de méthode d'une instruction **with** est interprété, si c'est possible, comme un membre de l'objet ou de l'enregistrement spécifié. Pour désigner une autre variable ou méthode portant le même nom que celui auquel vous accédez avec l'instruction **with**, vous devez le préfixer avec un qualificateur comme dans l'exemple suivant :

```
with DateCommande do
  begin
    Annee := Unit1.Annee
    ⋮
  end;
```

Quand plusieurs objets ou enregistrements apparaissent après le mot réservé **with**, l'instruction est traité comme une série d'instructions **with** imbriquée. Ainsi

with obj_1, obj_2, ..., obj_n do *instruction*

est équivalent à

with obj_1 do
  with obj_2 do
    ⋮
    with obj_n do
      instruction

Dans ce cas, chaque référence de variable ou nom de méthode de *instruction* est interprété, si c'est possible, comme un membre de obj_n ; sinon, il est interprété, si c'est possible, comme un membre de obj_{n-1}; et ainsi de suite. La même règle s'applique pour l'interprétation même des *objs* : si obj_n est un membre de obj_1 et de obj_2, il est interprété comme $obj_2.obj_n$.

Instructions If

L'instruction **if** a deux formes : **if...then** et **if...then...else**. La syntaxe de l'instruction **if...then** est :

if *expression* then *instruction*

où *expression* renvoie une valeur booléenne. Si *expression* vaut *True*, alors *instruction* est exécutée ; sinon elle ne l'est pas. Par exemple :

```
if J <> 0 then Resultat := I/J;
```

La syntaxe de l'instruction **if...then...else** est :

if *expression* then $instruction_1$ else $instruction_2$

où *expression* renvoie une valeur booléenne. Si *expression* vaut *True*, alors $instruction_1$ est exécutée ; sinon $instruction_2$ est exécutée.

Par exemple :

```
if J = 0 then
  Exit
else
  Resultat := I/J;
```

Les clauses **then** et **else** contiennent une seule instruction chacune, mais ce peut être une instruction structurée.Par exemple :

```
if J <> 0 then
begin
  Resultat := I/J;
  Compteur := Compteur + 1;
end
else if Compteur = Fin then
  Arret := True
else
  Exit;
```

Remarquez qu'il n'y a jamais de point-virgule entre la clause **then** et le mot **else**. Vous pouvez placer un point-virgule après une instruction **if** pour la séparer de l'instruction suivante du bloc mais les clauses **then** et **else** ne nécessitent rien d'autre qu'un espace ou un passage à la ligne entre elles. Le fait de placer un point-virgule immédiatement avant le **else** (dans une instruction **if**) est une erreur de programmation courante.

Un problème particulier se présente quand des instructions **if** sont imbriquées. Le problème se pose car certaines instructions **if** ont une clause **else** alors que d'autres ne l'ont pas, mais la syntaxe des deux variétés de l'instruction est pour le reste la même. Dans une série de conditions imbriquées où il y a moins de clauses **else** que d'instructions **if**, il n'est pas toujours évident de savoir à quel **if** une clause **else** est rattachée. Soit une instruction de la forme

`if` $expression_1$ `then if` $expression_2$ `then` $instruction_1$ `else` $instruction_2$;

Il y a deux manières d'analyser cette instruction :

`if` $expression_1$ `then` [`if` $expression_2$ `then` $instruction_1$ `else` $instruction_2$];

`if` $expression_1$ `then` [`if` $expression_2$ `then` $instruction_1$] `else` $instruction_2$;

Le compilateur analyse toujours de la première manière. C'est-à-dire que dans du véritable code, l'instruction :

```
if ... { expression1 } then
  if ... { expression2 } then
    ... { instruction1 }
  else
    ... { instruction2 } ;
```

est équivalent à :

```
if ... { expression1 } then
begin
  if ... { expression2 } then
    ... { instruction1 }
  else
```

```
    ... { instruction2 }
end;
```

La règle veut que les conditions imbriquées sont analysées en partant de la condition la plus interne, chaque **else** étant lié au plus proche **if** disponible à sa gauche. Pour forcer le compilateur à lire notre exemple de la deuxième manière, vous devez l'écrire explicitement de la manière suivante :

```
if ... { expression1 } then
begin
  if ... { expression2 } then
    ... { instruction1 }
end
else
  ... { instruction2 } ;
```

Instructions Case

L'instruction **case** propose une alternative plus lisible à l'utilisation de conditions **if** imbriquées complexes. Une instruction **case** a la forme

```
case expressionSelection of
  listeCas1: instruction1;
  ⋮
  listeCasn: instructionn;
end
```

où *expressionSelection* est une expression de type scalaire (les types chaîne sont interdits) et chaque *listeCas* est l'un des éléments suivants :

- Un nombre, une constante déclarée ou une expression que le compilateur peut évaluer sans exécuter le programme. Ce doit être une valeur de type scalaire compatible avec *expressionSelection*. Ainsi `7`, `True`, `4 + 5 * 3`, `'A'`, et `Integer('A')` peuvent être utilisés comme *listeCas*, mais les variables et la plupart des appels de fonctions ne peuvent être utilisés. Certaines fonctions prédéfinies comme *Hi* et *Lo* peuvent s'utiliser dans une *listeCas*$_1$. Voir "Expressions constantes" à la page 5-45.
- Un intervalle de la forme *Premier..Dernier*, où *Premier* et *Dernier* respectent tous les deux les critères précédents et où *Premier* est inférieur ou égal à *Dernier*.
- Une liste de la forme *élément*$_1$, ..., *élément*$_n$, où chaque *élément* respecte l'un des critères précédents.

Chaque valeur représentée par une *listeCas* doit être unique dans l'instruction **case** ; les intervalles et les listes ne peuvent se chevaucher. Une instruction **case peut avoir une clause else** finale :

```
case expressionSelection of
  listeCas1: instruction1;
  ⋮
  listeCasn: instructionn;
else
  instructions;
end
```

où instructions est un suite d'instructions séparées par des points virgules. Quand une instruction **case** est exécutée, au moins l'une des *instruction*$_1$ ... *instruction*$_n$ est exécutée. Le *listeCas* dont la valeur est égale à celle de *expressionSelection* détermine l'*instruction* à utiliser. Si aucun des *listeCas* n'a la même valeur que *expressionSelection*, alors ce sont les instructions de la clause **else** (s'il y en a) qui sont exécutées.

L'instruction **case** :

```
case I of
  1..5: Caption := 'Bas';
  6..9: Caption := 'Baut';
  0, 10..99: Caption := 'Hors de l''intervalle';
else
  Caption := '';
end;
```

est équivalente à la condition imbriquée suivante :

```
if I in [1..5] then
  Caption := 'Bas'
  else if I in [6..10] then
    Caption := 'Haut'
    else if (I = 0) or (I in [10..99]) then
      Caption := 'Hors de l''intervalle'
      else
        Caption := '';
```

Voici d'autres exemples d'instructions **case** :

```
case MaCouleur of
  Rouge: X := 1;
  Vert: X := 2;
  Bleu: X := 3;
  Jaune, Orange, Noir: X := 0;
end;

case Selection of
  Fin: Form1.Close;
  Calcul: CalculTotal(CourUnit, Quant);
else
  Beep;
end;
```

Boucles de contrôle

Les boucles permettent d'exécuter de manière répétitive une suite d'instructions en utilisant une condition ou une variable de contrôle pour déterminer quand arrêter l'exécution de la boucle. Le Pascal Objet dispose de trois sortes de boucles de contrôle : **les instructions repeat**, les instructions **while et les instructions for**.

Vous pouvez utiliser les procédures standard *Break* et *Continue* pour contrôler le flux des instructions **repeat**, **while** ou **for**. *Break* arrête l'instruction dans laquelle elle se produit alors que *Continue* fait passer l'exécution à l'itération suivante de la séquence. Pour davantage d'informations sur ces procédures, voir l'aide en ligne.

Instructions repeat

L'instruction **repeat** a la syntaxe suivante :

repeat *instruction*$_1$; ...; *instruction*$_n$; until *expression*

où *expression* renvoie une valeur booléenne. Le dernier point-virgule avant **until** est facultatif. L'instruction **repeat** exécute répétitivement la séquence d'instructions qu'elle contient en testant *expression* à chaque itération. Quand *expression* renvoie *True*, l'instruction **repeat** s'arrête. La séquence est toujours exécutée au moins une fois car *expression* n'est évaluée qu'après la première itération.

Voici des exemples d'instructions **repeat**

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Entrez une valeur (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

Instructions while

Une instruction **while** est similaire à l'instruction **repeat** à cette différence près que la condition de contrôle est évaluée avant la première itération de la séquence d'instructions. Donc si la condition est fausse, la séquence d'instructions n'est jamais exécutée.

L'instruction **while** a la syntaxe suivante :

while *expression* do *instruction*

où *expression* renvoie une valeur booléenne et *instruction* peut être une instruction composée. L'instruction **while** exécute répétitivement son *instruction*, en testant *expression* avant chaque itération. Tant que *expression* renvoie *True*, l'exécution se poursuit.

Voici des exemples d'instructions **while**

```
while Data[I] <> X do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(FicSource) do
begin
  Readln(FicSource, Ligne);
  Process(Ligne);
end;
```

Instructions for

Une instruction **for**, à la différence des instructions **repeat** et **while**, nécessite la spécification explicite du nombre d'itérations que la boucle doit effectuer. L'instruction **for** a la syntaxe suivante :

```
for compteur := valeurInitiale to valeurFinale do instruction
```

ou

```
for compteur := valeurInitiale downto valeurFinale do instruction
```

où

- *compteur* est une variable locale (déclarée dans le bloc contenant l'instruction **for**) de type scalaire sans aucun qualificateur.
- *valeurInitiale* et *valeurFinale* sont des expressions compatibles pour l'affectation avec *compteur*.
- *instruction* est une instruction simple ou structurée qui ne modifie pas la valeur de *compteur*.

L'instruction **for** affecte la valeur *valeurInitiale* à *compteur*, puis exécute répétitivement *instruction*, en incrémentant ou en décrémentant *compteur* après chaque itération. La syntaxe **for...to** incrémente *compteur* alors que la syntaxe **for...downto** le décrémente. Quand *compteur* renvoie la même valeur que *valeurFinale*, *l'instruction* est exécutée une dernière fois puis l'instruction **for** s'arrête. En d'autres termes, *instruction* est exécutée une fois pour chaque valeur de l'intervalle allant de *valeurInitiale* à *valeurFinale*. Si *valeurInitiale* est égale à *valeurFinale*, *instruction* est exécutée une seule fois. Si *valeurInitiale* est supérieure à *valeurFinale* dans une instruction **for...to** ou inférieure ou égale à *valeurFinale* dans une instruction **for...downto**, alors l'*instruction* n'est jamais exécutée. Après l'arrêt de l'instruction **for**, la valeur de *compteur* est non définie.

Afin de contrôler l'exécution de la boucle, la valeur des expressions *valeurInitiale* et *valeurFinale* n'est évaluée qu'une seule fois, avant le commencement de la boucle. Donc, une instruction **for...to** est presque identique à la construction **while** suivante :

```
begin
  compteur := valeurInitiale;
  while compteur <= valeurFinale do
  begin
    instruction;
    compteur := Succ(compteur);
  end;
end
```

La différence entre cette construction et l'instruction **for...to** est que la boucle **while** réévalue *valeurFinale* avant chaque itération. Cela peut réduire la vitesse d'exécution de manière sensible si *valeurFinale* est une expression complexe. De plus, cela signifie qu'une modification de la valeur *valeurFinale* dans *instruction* peut affecter l'exécution de la boucle.

Voici des exemples d'instructions **for** :

```
for I := 2 to 63 do
  if Donnees[I] > Max then
    Max := Donnees[I];

for I := ListBox1.Items.Count - 1 downto 0 do
      ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);

for I := 1 to 10 do
  for J := 1 to 10 do
  begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;

for C := Red to Blue do Verif(C);
```

Blocs et portée

Les déclarations et les instructions sont organisées en *blocs* qui définissent des noms de domaine locaux (ou *portées*) pour les labels et les identificateurs. Les blocs permettent à un même identificateur, par exemple un nom de variable, d'avoir des significations différentes dans différentes parties d'un programme. Chaque bloc fait partie de la déclaration d'un programme, d'une fonction ou d'une procédure ; la déclaration de chaque programme, fonction ou procédure est composée d'un seul bloc.

Blocs

Un bloc est composé d'une série de déclarations suivies d'une instruction composée. Toutes les déclarations doivent se trouver rassemblées au début du bloc. Un bloc a donc la forme suivante :

déclarations
begin
  instructions
end

La section *déclarations* peut contenir, dans un ordre quelconque, des déclarations de variables, de constantes (y compris des chaînes de ressource), de types, de procédures, de fonctions et de labels. Dans un bloc de programme, la section *déclarations* peut également contenir une ou plusieurs clauses **exports** (voir chapitre 9, “Bibliothèques et paquets”).

Par exemple, dans la déclaration de fonction suivante :

```
function Majuscule (const S: string): string;
var
  Ch: Char;
  L: Integer;
```

```
  Source, Dest: PChar;
begin
  ⋮
end;
```

La première ligne de la déclaration est l'en-tête de fonction et toutes les lignes suivantes constituent le bloc de la fonction. *Ch*, *L*, *Source* et *Dest* sont des variables locales ; leur déclaration n'est valable que dans le bloc de la fonction *Majuscule et* redéfinit (uniquement dans ce bloc) toute déclaration des mêmes identificateurs faite dans le bloc du programme ou dans les sections interface ou implémentation d'une unité.

Portée

Un identificateur, par exemple une variable ou un nom de fonction, ne peut être utilisé qu'à l'intérieur de la *portée* de sa déclaration. L'emplacement d'une déclaration détermine sa portée. La portée d'un identificateur déclaré dans la déclaration d'un programme, d'une fonction ou d'une procédure est limitée au bloc dans lequel il a été déclaré. Un identificateur déclaré dans la section interface d'une unité a une portée qui inclut toutes les autres unités et programmes utilisant l'unité où cette déclaration est faite. Les identificateurs ayant une portée plus restreinte (en particulier les identificateurs déclarés dans les fonctions et procédures) sont parfois dits *locaux* alors que les identificateurs ayant une portée plus étendue sont appelée *globaux*.

Les règles déterminant la portée d'un identificateur sont résumées ci-dessous :

Si l'identificateur est déclaré dans ...	Sa portée s'étend ...
La déclaration d'un programme, d'une fonction ou d'une procédure.	Depuis le point où il a été déclaré jusqu'à la fin du bloc en cours, y compris tous les blocs inclus dans cette portée.
La section interface d'une unité.	Depuis le point où il a été déclaré jusqu'à la fin de l'unité et dans toutes les unités ou programmes utilisant cette unité. Voir chapitre 3, "Programmes et unités".)
La section implémentation d'une unité mais hors du bloc d'une fonction ou d'une procédure.	Depuis le point où il a été déclaré jusqu'à la fin de la section implémentation. L'identificateur est disponible dans toutes les fonctions et procédures de la section implémentation.
La définition d'un type enregistrement (c'est-à-dire que l'identificateur est le nom d'un champ de l'enregistrement).	Depuis le point où il a été déclaré jusqu'à la fin de la définition du type de champ. Voir "Enregistrements" à la page 5-23.)
La définition d'une classe (c'est-à-dire que l'identificateur est le nom d'une propriété ou d'une méthode de la classe).	Depuis le point où il a été déclaré jusqu'à la fin de la définition du type classe et également dans les définitions des descendants de la classe et les blocs de toutes les méthodes de la classe et de ses descendants. Voir chapitre 7, "Classes et objets".

Conflits de nom

Quand un bloc en comprend un autre, le premier est appelé *bloc extérieur* et l'autre est appelé *bloc intérieur*. Si un identificateur déclaré dans le bloc extérieur est redéclaré dans le bloc intérieur, la déclaration intérieure redéfinit l'extérieure et détermine la signification de l'identificateur pour la durée du bloc intérieur. Si, par exemple, vous avez déclaré une variable appelée *ValeurMax* dans la section interface d'une unité, puis si vous déclarez une autre variable de même nom dans une déclaration de fonction de cette unité, toute occurrence non qualifiée de *ValeurMax* dans le bloc de la fonction est régit par la deuxième définition, celle qui est locale. De même, une fonction déclarée à l'intérieur d'une autre fonction crée une nouvelle portée interne dans laquelle les identificateurs utilisés par la fonction externe peuvent être localement redéfinis.

L'utilisation de plusieurs unités complique davantage la définition de portée. Chaque unité énumérée dans une clause **uses** impose une nouvelle portée qui inclut les unités restantes utilisées et le programme ou l'unité contenant la clause **uses**. La première unité d'une clause **uses** représente la portée la plus externe, et chaque unité successive représente une nouvelle portée interne à la précédente. Si plusieurs unités déclarent le même identificateur dans leur section interface, une référence sans qualificateur à l'identificateur sélectionne la déclaration effectuée dans la portée la plus externe, c'est-à-dire dans l'unité où la référence est faite, ou, si cette unité ne déclare pas l'identificateur dans la dernière unité de la clause **uses** qui déclare cet identificateur.

L'unité *System* est utilisée automatiquement par chaque programme et unité. Les déclarations de *System* ainsi que les types prédéfinis, les routines et les constantes reconnues automatiquement par le compilateur ont toujours la portée la plus extérieure.

Vous pouvez redéfinir ces règles de portée et court-circuiter une déclaration intérieure en utilisant un identificateur qualifié (voir “Identificateurs qualifiés” à la page 4-3) ou une instruction **with** (voir “Instructions With” à la page 4-23).

Chapitre

5

Types de données, variables et constantes

Un *type* est essentiellement le nom d'une sorte de données. Quand vous déclarez une variable, il faut spécifier son type qui détermine l'ensemble des valeurs que la variable peut contenir et les opérations qui peuvent être effectuées. Chaque expression renvoie des données d'un type particulier tout comme les fonctions. La plupart des fonctions et des procédures nécessitent des paramètres ayant un type spécifique.

Le Pascal Objet est un langage "fortement typé" : cela signifie qu'il distingue un grand nombre de types de données et ne permet pas toujours de substituer un type de données à un autre. C'est généralement un avantage car cela permet au compilateur de traiter les données intelligemment et de vérifier plus précisément votre code, ce qui évite des erreurs d'exécution difficiles à diagnostiquer. Toutefois, si vous avez besoin d'une plus grande flexibilité, il est possible de contourner la vérification stricte des types. Les solutions possibles sont le *transtypage* (voir "Transtypage" à la page 4-16), les *pointeurs* (voir "Pointeurs et types pointeur" à la page 5-27), les *variants* (voir "Types variants" à la page 5-33), *les parties variables* dans les enregistrements (voir "Partie variable d'enregistrements" à la page 5-24), et l'*adressage absolu des* variables (voir "Adresses absolues" à la page 5-42).

A propos des types

Il y a plusieurs manières de regrouper les types de données du Pascal Objet :

- Certains types sont *prédéfinis* ; le compilateur les reconnaît automatiquement sans nécessiter une déclaration. La plupart des types décrits dans cette présentation du langage sont des types prédéfinis. D'autres types sont créés à l'aide de déclarations ; c'est le cas des types définis par l'utilisateur et de ceux définis dans les bibliothèques du produit.

- Les types peuvent se diviser en types *fondamentaux* et en types *génériques*. L'étendue et le format d'un type fondamental sont les mêmes dans toutes les implémentations du Pascal Objet, indépendamment de la CPU utilisée ou du système d'exploitation. L'étendue et le format d'un type générique sont spécifiques à une plate-forme et peuvent varier selon les implémentations. La plupart des types prédéfinis sont fondamentaux mais certains types d'entiers, de caractères, de chaînes et de pointeurs sont génériques. Il est préférable d'utiliser, dans la mesure du possible, les types génériques car ils permettent des performances optimales et la portabilité du code. Cependant, la modification du format de stockage selon l'implémentation d'un type générique peut poser des problèmes de compatibilité, par exemple si vous placez les données dans un fichier.
- Les types peuvent aussi se diviser en types *simple*, *chaîne*, *structuré*, *pointeur*, *procédure* ou *variant*. De plus, les identificateurs de type même peuvent être considérés comme appartenant à un “type” particulier car ils sont transmis comme paramètres dans certaines fonctions (par exemple *High*, *Low* ou *SizeOf*).

Le schéma suivant décrit la taxonomie des types de données Pascal Objet.

simple
- scalaire
  - entier
  - caractère
  - booléen
  - énuméré
  - intervalle
- réel

chaîne

structuré
- ensemble
- tableau
- enregistrement
- fichier
- classe
- référence de classe
- interface

pointeur

procédure

variant

identificateur de type

La fonction standard *SizeOf* opère sur toutes les variables et tous les identificateurs de type. Elle renvoie un entier représentant la quantité de mémoire (en octets) utilisée pour stocker les données du type spécifié. Par exemple, `SizeOf(Longint)` renvoie 4, car une variable *Longint* utilise quatre octets de mémoire.

Les déclarations de type sont décrites dans les sections suivantes. Pour des informations générales sur la déclaration de type, voir “Déclaration de types” à la page 5-40.

Types simples

Les types simples qui comportent les types *scalaire* et *réel*, définissent des ensembles ordonnés de valeurs.

Types scalaires

Les types scalaires sont les types *entier*, *caractère*, *booléen*, *énuméré* et *intervalle*. Un type scalaire définit un ensemble ordonné de valeurs dont chaque valeur, sauf la première, a un *prédécesseur* unique et dont chaque valeur, sauf la dernière, a un *successeur unique*. Chaque valeur a un *rang* qui détermine l'ordre du type. Dans la plupart des cas, si une valeur a le rang n, son prédécesseur a le rang $n-1$ et son successeur a le rang $n+1$.

- Pour les entiers, le rang d'une valeur est la valeur elle-même.
- Les types intervalle conservent les rangs de leurs types de base.
- Pour les autres types scalaires, la première valeur a par défaut le rang 0, la suivante a le rang 1, etc. La déclaration d'un type énuméré doit explicitement redéfinir la valeur par défaut.

Plusieurs fonctions prédéfinies portent sur des valeurs scalaires et les identificateurs de type. Les plus importantes sont résumées ci-dessous.

Fonction	Paramètre	Valeur renvoyée	Remarque
Ord	expression scalaire	rang de la valeur de l'expression	Ne prend pas d'arguments *Int64*.
Pred	expression scalaire	prédécesseur de la valeur de l'expression	Ne pas utiliser sur les propriétés ayant une procédure **write**.
Succ	expression scalaire	successeur de la valeur de l'expression	Ne pas utiliser sur les propriétés ayant une procédure **write**.
High	identificateur de type scalaire ou variable de type scalaire	plus grande valeur du type	Opère également sur les types de chaîne courte et les tableaux.
Low	identificateur de type scalaire ou variable de type scalaire	plus petite valeur du type	Opère également sur les types de chaîne courte et les tableaux.

Par exemple, `High(Byte)` renvoie 255 car la plus grande valeur du type *Byte* est 255, et `Succ(2)` renvoie 3 car 3 est le successeur de 2.

Les procédures standard *Inc* et *Dec* incrémentent et décrémentent la valeur d'une variable scalaire. Par exemple, `Inc(I)` est équivalent à `I := Succ(I)` et, si *I* est une variable entière, c'est également équivalent à `I := I + 1`.

Types entiers

Un type entier représente un sous-ensemble des nombres. Les types entiers génériques sont *Integer* et *Cardinal* ; utilisez-les dans la mesure du possible car ils donnent de meilleures performances avec la CPU et le système d'exploitation utilisés. Le tableau suivant spécifie leur étendue et leur format de stockage pour la version actuelle du compilateur 32 bits Pascal Objet.

Tableau 5.1 Types entiers génériques pour l'implémentation 32 bits du Pascal Objet

Type	Etendue	Format
Integer	–2147483648..2147483647	32 bits signé
Cardinal	0..4294967295	32 bits non signé

Les types fondamentaux signés sont *Shortint*, *Smallint*, *Longint*, *Int64*, *Byte*, *Word* et *Longword*.

Tableau 5.2 Types entiers fondamentaux

Type	Etendue	Format
Shortint	–128..127	8 bits signé
Smallint	–32768..32767	16 bits signé
Longint	–2147483648..2147483647	32 bits signé
Int64	$-2^{63}..2^{63}-1$	64 bits signé
Byte	0..255	8 bits non signé
Word	0..65535	16 bits non signé
Longword	0..4294967295	32 bits non signé

En général, les opérations arithmétiques sur les entiers renvoient une valeur de type *Integer*, qui pour l'implémentation actuelle est équivalente au type *Longint sur 32 bits*. Les opérations ne renvoient une valeur de type *Int64* que si elles portent sur un opérande *Int64*. Par exemple, le code suivant donne des résultats incorrects.

```
var
  I: Integer;
  J: Int64;
  ⋮
I := High(Integer);
J := I + 1;
```

Pour obtenir une valeur renvoyée de type *Int64*, convertissez *I* en *Int64* :

```
 ⋮
J := Int64(I) + 1;
```

Pour davantage d'informations, voir “Opérateurs arithmétiques” à la page 4-7.

Remarque La plupart des routines standard qui attendent des arguments entiers tronquent les valeurs *Int64* à 32 bits. Néanmoins les routines *High*, *Low*, *Succ*, *Pred*, *Inc*, *Dec*, *IntToStr* et *IntToHex* gèrent pleinement les arguments *Int64*. De même, les fonctions *Round*, *Trunc*, *StrToInt64* et *StrToInt64Def* renvoient des valeurs *Int64*. Quelques routines (dont *Ord)* n'acceptent pas les valeurs *Int64*.

Si vous incrémentez la dernière valeur ou si vous décrémentez la première valeur d'un type entier, le résultat boucle sur le début ou la fin de l'étendue. Par exemple, le type *Shortint* a l'étendue –128..127 ; donc après l'exécution du code suivant :

```
var I: Shortint;
 ⋮
I := High(Shortint);
I := I + 1;
```

I a la valeur –128. Si la vérification des limites de compilation est activée, ce code génère néanmoins une erreur d'exécution.

Types caractère

Les types de caractère fondamentaux sont *AnsiChar* et *WideChar*. Les valeurs *AnsiChar* sont des caractères sur un octet (8 bits) ordonnés selon le jeu de caractères local qui est peut-être multi-octets. *AnsiChar* a été initialement conçu à partir du jeu de caractères ANSI (d'où son nom), mais a été maintenant étendu pour s'adapter au jeu de caractères de la locale en cours.

Les caractères *WideChar* utilisent plus d'un octet pour représenter chaque caractère. Les valeurs *WideChar* sont des caractères sur un mot (16 bits) ordonnés selon le jeu de caractères Unicode (ils pourraient être plus longs dans de futures implémentations). Les 256 premiers caractères Unicode correspondent aux caractères ANSI.

Le type de caractère générique *Char* est équivalent à *AnsiChar*. Comme l'implémentation de *Char* est susceptible de changer, il est judicieux d'utiliser la fonction standard *SizeOf* plutôt qu'une constante codée en dur dans les programmes qui doivent gérer des caractères de tailles différentes.

Une constante chaîne de longueur 1, comme `'A'`, peut désigner une valeur caractère. La fonction prédéfinie *Chr* renvoie la valeur caractère pour tout entier dans l'étendue de *AnsiChar* ou de *WideChar* ; ainsi, `Chr(65)` renvoie la lettre *A*.

Les valeurs de caractère, comme les entiers, bouclent quand elles sont décrémentées ou incrémentées au-delà du début ou de la fin de leur étendue (à moins que la vérification des limites ne soit activée). Ainsi, une fois le code suivant exécuté :

```
var
  Lettre: Char;
  I: Integer;
begin
  Lettre := High(Lettre);
  for I := 1 to 66 do
    Inc(Lettre);
end;
```

Lettre a la valeur *A* (ASCII 65).

Pour davantage d'informations sur les caractères Unicode, voir "A propos des jeux de caractères étendus" à la page 5-14 et "Manipulation des chaînes à zéro terminal" à la page 5-14.

Types booléens

Les quatre type booléens prédéfinis sont *Boolean*, *ByteBool*, *WordBool* et *LongBool*. *Boolean* est le type de prédilection. Les autres existent uniquement pour proposer une compatibilité avec d'autres langages et systèmes d'exploitation.

Une variable *Boolean* occupe un octet de mémoire, une variable *ByteBool* occupe également un octet, une variable *WordBool* occupe deux octets (un mot), et une variable *LongBool* occupe quatre octets (deux mots).

Les valeurs booléennes sont désignées par les constantes prédéfinies *True* et *False*. Les relations suivantes s'appliquent.

Boolean	ByteBool, WordBool, LongBool
False < *True*	*False* <> *True*
Ord(*False*) = 0	*Ord*(*False*) = 0
Ord(*True*) = 1	*Ord*(*True*) <> 0
Succ(*False*) = *True*	*Succ*(*False*) = *True*
Pred(*True*) = *False*	*Pred*(*False*) = *True*

Une valeur de type *ByteBool*, *LongBool* ou *WordBool* est considérée comme *True* quand son rang est non nul. Si une telle valeur apparaît dans un contexte où un *Boolean* est attendu, le compilateur convertit automatiquement toute valeur de rang non nul en *True*.

La remarque précédente porte sur le rang des valeurs booléennes, non pas sur les valeurs mêmes. En Pascal Objet, les expressions booléennes ne peuvent être comparées avec des entiers ou des réels. Par exemple, si *X* est une variable entière, l'instruction :

```
if X then ...;
```

génère une erreur de compilation. Le transtypage de la variable en un type booléen n'est pas fiable mais les deux solutions suivantes sont utilisables.

```
if X <> 0 then ...; { utilise une expression plus longue qui renvoie une valeur booléenne }

var OK: Boolean     { utilise une variable booléenne }
 ⋮
if X <> 0 then OK := True;
if OK then ...;
```

Types énumérés

Un type énuméré définit un ensemble ordonné de valeurs simplement en énumérant les identificateurs désignant ces valeurs. Les valeurs n'ont pas de signification propre. Pour déclarer un type énuméré, utilisez la syntaxe suivante :

type *nomType* = (val_1, ..., val_n)

où *nomType* et les *val* sont des identificateurs valides. Par exemple, la déclaration :

```
type Suite = (Trefle, Carreau, Coeur, Pique);
```

définit une suite énumérée appelée *Suite* dont les valeurs possibles sont *Trefle*, *Carreau*, *Coeur* et *Pique*, où `Ord(Trefle)` renvoie 0, `Ord(Carreau)` renvoie 1, etc.

Quand vous déclarez un type énuméré, vous déclarez chaque *val* comme une constante de type *nomType*. Si les identificateurs *val* sont utilisés dans un autre but dans la même portée, il y a un conflit de nom. Si, par exemple vous déclarez le type :

```
type TSon = (Click, Clack, Clock);
```

Mais *Click* est également le nom d'une méthode définie dans *TControl* et dans tous les objets de la VCL et/ou CLX qui en dérivent. Donc, si vous écrivez une application et écrivez un gestionnaire d'événement de la forme suivante :

```
procedure TForm1.DBGrid1Enter(Sender: TObject);
var Truc: TSon;
begin
  ⋮
  Truc := Click;
  ⋮
end;
```

Vous obtenez une erreur de compilation ; le compilateur interprète *Click* dans la portée de la procédure comme une référence à la méthode *Click de TForm*. Vous pouvez contourner ce problème en qualifiant l'identificateur : si *TSon* est déclarée dans *MonUnit*, vous devez utiliser :

```
Truc := MonUnit.Click;
```

Une meilleure solution est néanmoins de choisir des noms de constante qui ne risquent pas de rentrer en conflit avec d'autres identificateurs. Par exemple :

```
type
  TSon = (tsClick, tsClack, tsClock);
  TMaCouleur = (mcRouge, mcBleu, mcVert, mcJaune, mcOrange);
  Reponse = (repOui, repNon, RepBof);
```

Vous pouvez utiliser directement la construction (val_1, ..., val_n) dans une déclaration de variable comme nom de type :

```
var MaCarte: (Trefle, Carreau, Coeur, Pique);
```

Mais si vous déclarez *MaCarte* de cette manière, vous ne pouvez déclarer d'autres variables dans la même portée en utilisant ces identificateurs de constantes. Ainsi :

```
var Carte1: (Trefle, Carreau, Coeur, Pique);
var Carte2: (Trefle, Carreau, Coeur, Pique);
```

génère une erreur de compilation. Mais :

```
var Carte1, Carte2: (Trefle, Carreau, Coeur, Pique);
```

se compile sans problème, tout comme :

```
type Suite = (Trefle, Carreau, Coeur, Pique);
var
  Carte1: Suite;
  Carte2: Suite;
```

Types énumérés dont les rangs sont attribués explicitement

Par défaut, le rang des valeurs énumérées commence à 0 et suit l'ordre des indentificateurs de la déclaration de type. Vous pouvez redéfinir cela en attribuant explicitement les rangs de certaines ou de toutes les valeurs de la déclaration. Pour attribuer un rang à une valeur, faites suivre son identificateur de = *expressionConstante*, où *expressionConstante* est une expression constante dont le résultat est un entier. (Voir “Expressions constantes” à la page 5-45.) Par exemple,

```
type Taille = (Petit = 5, Moyen = 10, Grand = Petit + Moyen);
```

définit un type appelé *Taille* dont les valeurs possibles sont *Petit*, *Moyen* et *Grand*, où `Ord(Petit)` renvoie 5, `Ord(Moyen)` renvoie 10 et `Ord(Grand)` renvoie 15.

Un type énuméré est en effet un intervalle dont les valeurs inférieure et supérieure correspondent aux rangs inférieur et supérieur des constantes de la déclaration. Dans l'exemple précédent, le type *Taille* a 11 valeurs possibles dont le rang va de 5 à 15. (Ici, le type `array[Taille] of Char` représente un tableau de 11 caractères.) Seules trois de ces valeurs ont un nom, mais les autres sont accessibles par le biais de conversions de types et de routines comme *Pred*, *Succ*, *Inc* et *Dec*. Dans l'exemple suivant, des valeurs “anonymes” dans l'étendue de *Taille* sont attribuées à la variable *X*.

```
var X: Taille;
X := Petit;    // Ord(X) = 5
X := Taille(6);  // Ord(X) = 6
Inc(X);        // Ord(X) = 7
```

Toute valeur à laquelle un rang n'a pas été attribué de façon explicite a pour rang un de plus que celui de la valeur précédente de la liste. Si aucun rang n'est attribué à la première valeur, son rang est 0. Ici, étant donné la déclaration

```
type UneEnum = (e1, e2, e3 = 1);
```

UneEnum prend seulement deux valeurs possibles : `Ord(e1)` renvoie 0, `Ord(e2)` renvoie 1, et `Ord(e3)` renvoie également 1 ; comme *e2* et *e3* ont le même rang, ils représentent la même valeur.

Types intervalle

Un type intervalle représente un sous-ensemble de valeurs d'un autre type (appelé le *type de base*). Toute construction de la forme *Bas..Haut*, où *Bas* et *Haut* sont des expressions constantes du même type scalaire, *Bas étant inférieur à Haut*, identifie un type intervalle qui inclut toutes les valeurs comprises entre *Bas* et *Haut*. Si par exemple, vous déclarez le type énuméré :

```
type TCouleurs = (Rouge, Bleu, Vert, Jaune, Orange, Violet, Blanc, Noir);
```

vous pouvez définir le type intervalle suivant :

```
type TMesCouleurs = Vert..Blanc;
```

Ici, *TMesCouleurs* inclut les valeurs *Vert*, *Jaune*, *Orange*, *Violet* et *Blanc*.

Vous pouvez utiliser des constantes numériques ou caractère (des constantes chaîne de longueur 1) pour définir des types intervalles :

```
type
  DesNombres = -128..127;
  Majs = 'A'..'Z';
```

Quand vous utilisez des constantes numérique ou caractère pour définir un intervalle, le type de base est le plus petit type entier ou caractère contenant l'intervalle spécifié.

La construction *Bas..Haut* fonctionne directement comme nom de type, vous pouvez donc l'utiliser directement dans des déclarations de variables. Par exemple ;

```
var DesNombres: 1..500;
```

déclare une variable entière dont la valeur est dans l'intervalle allant de 1 à 500.

Le rang de chaque valeur d'un intervalle est celui qu'elle a dans le type de base. Dans le premier exemple, si *Couleur* est une variable contenant la valeur *Vert*, `Ord(Couleur)` renvoie 2, que *Couleur* soit de type *TCouleurs* ou *TMesCouleurs*. Les valeurs ne bouclent pas au début ou à la fin de l'intervalle même si le type de base est de type entier ou caractère ; l'incrémentation ou la décrémention au-delà des limites d'un intervalle convertit simplement la valeur dans le type de base. Ainsi :

```
type Pourcentage = 0..99;
var I: Pourcentage;
⋮
I := 100;
```

produit une erreur alors que :

```
⋮
I := 99;
Inc(I);
```

affecte la valeur 100 à *I* (à moins que la vérification des limites de compilation ne soit activée).

L'utilisation d'expressions constantes dans une définition d'intervalle introduit une difficulté syntaxique. Dans toute déclaration de type, quand le premier caractère significatif après le = est une parenthèse gauche, le compilateur suppose qu'un type énuméré est défini. Ainsi, le code :

```
const
  X = 50;
  Y = 10;
type
  Echelle = (X - Y) * 2..(X + Y) * 2;
```

produit une erreur. Pour contourner ce problème, il faut réécrire la déclaration de type afin d'éviter la parenthèse de début :

```
type
  Echelle = 2 * (X - Y)..(X + Y) * 2;
```

Types réels

Un type réel définit un ensemble de nombres pouvant être représentés par une notation à virgule flottante. Le tableau suivant donne l'étendue et le format de stockage des types réels fondamentaux.

Tableau 5.3 Types réels fondamentaux

Type	Etendue	Chiffres significatifs	Taille en octets
Real48	2.9×10^{-39} .. 1.7×10^{38}	11–12	6
Single	1.5×10^{-45} .. 3.4×10^{38}	7–8	4
Double	5.0×10^{-324} .. 1.7×10^{308}	15–16	8
Extended	3.6×10^{-4951} .. 1.1×10^{4932}	19–20	10
Comp	$-2^{63}+1$ .. $2^{63}-1$	19–20	8
Currency	–922337203685477.5808.. 922337203685477.5807	19–20	8

Le type générique *Real* est équivalent, dans son implémentation actuelle, au type *Double*.

Tableau 5.4 Type réel générique

Type	Etendue	Chiffres significatifs	Taille en octets
Real	5.0×10^{-324} .. 1.7×10^{308}	15–16	8

Remarque Le type *Real48* sur six octets s'appelait *Real* dans les versions précédentes du Pascal Objet. Si vous recompilez du code utilisant ce type *Real* sur six octets ancienne manière, vous pouvez le changer en *Real48*. Vous pouvez également utiliser la directive de compilation **{$REALCOMPATIBILITY ON}** qui revient à l'interprétation de *Real* comme un type sur six octets.

Les remarques suivantes s'appliquent aux types réels fondamentaux.

- *Real48* est conservé pour la compatibilité ascendante. Comme son format de stockage n'est pas géré naturellement par les processeurs Intel, ce type produit des performances plus mauvaises que les autres types à virgule flottante.
- *Extended* propose une meilleure précision que les autres types réels, mais il est moins portable. Evitez d'utiliser *Extended* si vous créez des fichiers de données qui doivent être partagés sur plusieurs plates-formes.
- Le type *Comp* est un type natif des processeurs Intel et représente un entier sur 64 bits. Il est néanmoins classé parmi les réels car il ne se comporte pas comme un type scalaire. Par exemple, il n'est pas possible d'incrémenter ou de décrémenter une valeur *Comp*. *Comp* est conservé uniquement pour la compatibilité ascendante. Utilisez le type *Int64* pour de meilleures performances.
- *Currency* est un type de données à virgule fixe qui limite les erreurs d'arrondi dans les calculs monétaires. Il est stocké dans un entier sur 64 bits, les quatre chiffres les moins significatifs représentant implicitement les chiffres après la

virgule. Quand il est combiné avec d'autres types réels dans des affectations et des expressions, les valeurs *Currency* sont automatiquement divisées ou multipliées par 10000.

Types chaîne

Une chaîne représente une suite de caractères. Le Pascal Objet gère les types de chaîne prédéfinis suivants.

Tableau 5.5 Types de chaîne

Type	Longueur maximum	Mémoire nécessaire	Utilisation
ShortString	255 caractères	de 2 à 256 octets	Compatibilité ascendante
AnsiString	~2^{31} caractères	de 4 octets à 2Go	Caractère sur 8 bits (ANSI)
WideString	~2^{30} caractères	de 4 octets à 2Go	Caractères Unicode ; serveurs muli-utilisateurs et applications multilingues

AnsiString, appelé parfois *chaîne longue* est le type le mieux adapté aux utilisations les plus diverses.

Les types chaîne peuvent être combinés dans les affectations et les expressions ; le compilateur effectue automatiquement les conversions nécessaires. Par contre, les chaînes transmises par adresse (référence) aux fonctions et procédures (comme paramètres **var** ou **out**) doivent être du type approprié. Les chaînes peuvent être explicitement converties dans un type de chaîne différent (voir "Transtypage" à la page 4-16).

Le mot réservé **string** fonctionne comme un identificateur de type générique. Par exemple :

```
var S: string;
```

crée une variable *S* contenant une chaîne. Dans l'état par défaut **{$H+}**, le compilateur interprète **string** (quand il apparaît sans être suivi d'un crochet ouvrant) comme désignant *AnsiString*. Utilisez la directive **{$H–}** pour que **string** soit interprété comme désignant *ShortString*.

La fonction standard *Length* renvoie le nombre de caractères d'une chaîne. La procédure *SetLength* ajuste la longueur d'une chaîne. Pour davantage d'informations, voir l'aide en ligne.

Les comparaisons de chaînes sont définies par l'ordre des caractères dans les positions correspondantes. Avec des chaînes de longueurs différentes, chaque caractère de la chaîne la plus longue n'ayant pas de caractère correspondant dans la chaîne la plus courte prend la valeur "supérieur à". Par exemple, "AB" est supérieure à "A" ; c'est-à-dire que `'AB' > 'A'` renvoie *True*. Les chaînes de longueur nulle contiennent les valeurs les plus petites.

Il est possible d'indicer une variable chaîne comme un tableau. Si *S* est une variable chaîne et *i* une expression entière, *S*[*i*] représente le *i*ème caractère — ou, plus précisément, le *i*ème octet — de *S*. Pour les types *ShortString* et

AnsiString, `S[i]` est de type *AnsiChar* ; pour le type *WideString*, `S[i]` est de type *WideChar*. L'instruction `Machaine[2] := 'A';` affecte la valeur *A* au second caractère de *MaChaine*. Le code suivant utilise la fonction standard *UpCase* pour convertir *Machaine* en majuscules :

```
var I: Integer;
begin
  I := Length(MaChaine);
  while I > 0 do
  begin
    MaChaine[I] := UpCase(MaChaine[I]);
    I := I - 1;
  end;
end;
```

Attention en utilisant de cette manière des indices sur les chaînes car si vous écrivez au-delà de la fin de la chaîne, vous provoquez des violations d'accès. De même, évitez de transmettre comme paramètre **var** des indices sur des chaînes longues car cela produit du code inefficace.

Vous pouvez affecter à variable la valeur d'une constante chaîne ou de toute expression renvoyant une chaîne. La longueur de la chaîne change de manière dynamique lors de l'affectation. En voici des exemples :

```
MaChaine := 'Bonjour chez vous!';
MaChaine := 'Bonjour ' + 'chez vous';
MaChaine := MaChaine + '!';
MaChaine := ' ';              { espace }
MaChaine := '';               { chaîne vide}
```

Pour davantage d'informations, voir "Chaînes de caractères" à la page 4-5 et "Opérateurs de chaînes" à la page 4-10.

Chaînes courtes

Une chaîne courte (*ShortString)* compte de 0 à 255 caractères. Si la longueur d'un *ShortString* peut changer de manière dynamique, 256 octets de mémoire lui sont alloués statiquement. Le premier octet stocke la longueur de la chaîne, les 255 octets suivants sont disponibles pour les caractères. Si *S* est une variable *ShortString*, `Ord(S[0])` renvoie, comme `Length(S)`, la longueur de *S* ; l'affectation d'une valeur à `S[0]`, comme l'appel de *SetLength*, modifie la longueur de *S*. *ShortString* utilise des caractères ANSI sur 8 bits, ce type est conservé uniquement pour la compatibilité ascendante.

Pascal Objet gère des types de chaîne courte, des sous-types de *ShortString* dont la longueur maximum se situe entre 0 et 255 caractères. Ils sont simplement désignés par un nombre entre crochets suivant le mot réservé **string**. Par exemple :

```
var MaChaine: string[100];
```

crée une variable appelée *MaChaine* dont la longueur maximum est de 100 caractères.

Cela revient aux déclarations suivantes :

```
type CChaine = string[100];
var MaChaine: CChaine;
```

Les variables déclarées ainsi allouent uniquement la mémoire nécessaire au type, c'est-à-dire la longueur maximum plus un octet. Dans l'exemple précédent, *MaChaine* utilise 101 octets, à comparer aux 256 octets d'une variable ayant le type prédéfini *ShortString*.

Quand vous affectez une valeur à une variable chaîne courte, la chaîne est tronquée si elle dépasse la longueur maximum du type.

Les fonctions standard *High* et *Low* acceptent les identificateurs de type chaîne courte et les variables de ces types. *High* renvoie la longueur maximum du type de chaîne courte alors que *Low* renvoie zéro.

Chaînes longues

Le type *AnsiString*, appelé également *chaîne longue*, représente une chaîne allouée dynamiquement dont la longueur maximum n'est limitée que par la mémoire disponible. Il utilise des caractères ANSI sur 8 bits.

Une variable chaîne longue est un pointeur occupant quatre octets de mémoire. Quand la variable est vide (c'est-à-dire quand elle contient une chaîne de longueur nulle), le pointeur vaut **nil** et la chaîne n'utilise pas de mémoire supplémentaire. Quand la variable est non vide, elle pointe sur un bloc de mémoire alloué dynamiquement qui contient la valeur de la chaîne, un indicateur de longueur sur 32 bits et un compteur de référence sur 32 bits. Cette mémoire est allouée sur le tas mais sa gestion est entièrement automatique et ne nécessite pas de codage.

Comme les variables chaîne longue sont des pointeurs, plusieurs peuvent pointer sur la même valeur sans utiliser de mémoire supplémentaire. Le compilateur utilise ceci pour économiser les ressources et exécuter les affectations plus rapidement. A chaque fois qu'une chaîne longue est supprimée ou qu'une nouvelle valeur lui est affectée, le compteur de référence de l'ancienne chaîne (la valeur précédente de la variable) est décrémenté et le compteur de référence de la nouvelle valeur (s'il y en a une) est incrémenté. Si le compteur de référence d'une chaîne devient nul, sa mémoire est libérée. Ce processus est appelé *comptage de références*. Quand des indices sont utilisés pour modifier la valeur d'un seul caractère d'une chaîne, une copie de la chaîne est effectuée si, et seulement si, son compteur de référence est supérieur à un. On parle alors de sémantique *copie-par-écriture*.

Chaînes étendues

Le type *WideString* représente une chaîne allouée dynamiquement composée de caractères Unicode sur 16 bits. Par de nombreux points, ce type ressemble au type *AnsiString*.

Dans Win32, *WideString* est compatible avec le type COM *BSTR*. Les outils de développement Borland disposent de fonctionnalités de conversion de valeurs *AnsiString* en valeurs *WideString*, mais vous pouvez être amené à transtyper ou à convertir explicitement vos chaînes en *WideString*.

A propos des jeux de caractères étendus

Windows et Linux gèrent tous les deux des jeux de caractères sur *un octet* et des jeux *multi-octets* mais aussi *Unicode*. Avec un jeu de caractère sur un seul octet (SBCS), chaque octet d'une chaîne représente un seul caractère. Le jeu de caractères ANSI utilisé par de nombreux systèmes d'exploitation orientaux est mono-octet.

Dans les jeux de caractères multi-octets (MBCS), certains caractères sont représentés par un seul octet et d'autres par plusieurs. Le premier octet d'un caractère multi-octets est appelé *octet de tête*. En général, les 128 premiers caractères d'un jeu de caractère multi-octets correspondent aux caractères ASCII sur 7 bits et tout octet dont le rang est supérieur à 127 est l'octet de tête d'un caractère multi-octets. Seul les caractères mono-octet peuvent contenir la valeur nulle (#0). Les jeux de caractères multi-octets, en particulier les jeux de caractères sur deux octets (DBCS), sont fréquemment utilisés pour les langues asiatiques, alors que le jeu de caractères UTF-8 utilisé par Linux est un encodage multi-octets d'Unicode.

Dans le jeu de caractères Unicode, chaque caractère est représenté par deux octets. Une chaîne Unicode est donc une séquence non pas d'octets mais de mots de deux octets. Les caractères et les chaînes Unicode sont également appelés *caractères étendus* et *chaînes de caractères étendus*. Les 256 premiers caractères Unicode correspondent au jeu de caractère ANSI. Windows supporte Unicode (UCS-2). Linux supporte UCS-4, un sur-ensemble de UCS-2. Delphi/Kylix supporte UCS-2 sur les deux plates-formes.

Le Pascal Objet gère les caractères et les chaînes mono et multi-octets via les types *Char*, *PChar*, *AnsiChar*, *PAnsiChar* et *AnsiString*. L'indexation des chaînes multi-octets n'est pas fiable, puisque *S*[*i*] représente le *i*ème octet (et non nécessairement le *i*ème caractère) de *S*. Mais, les fonctions standard de manipulation de chaînes ont des équivalents multi-octets qui gèrent également l'ordre de tri des caractères spécifiques à la localisation. Le nom des fonctions portant sur les caractères multi-octets commencent généralement par *Ansi-*. Ainsi, la version multi-octets de *StrPos* est *AnsiStrPos*. La gestion des caractères multi-octets dépend du système d'exploitation et est basée sur la localisation en cours.

Le Pascal Objet gère les caractères et les chaînes Unicode via les types *WideChar*, *PWideChar* et *WideString*.

Manipulation des chaînes à zéro terminal

Beaucoup de langages de programmation, dont le C et le C++, n'ont pas de type de donnée chaîne naturel. Ces langages, et des environnements qui ont été construits avec, utilisent des chaînes à *zéro terminal*. Une chaîne à zéro terminal est un tableau de caractères d'indice de base zéro et terminé par NULL (#0).

Comme le tableau n'a pas d'indicateur de longueur, le premier caractère NULL indique la fin de la chaîne. Vous pouvez utiliser des constructions Pascal Objet et des routines spéciales de l'unité *SysUtils* (voir chapitre 8, "Routines standard et Entrées/Sorties") pour manipuler des chaînes à zéro terminal quand vous avez besoin de partager des données avec des systèmes les utilisant.

Par exemple, les déclarations de type suivantes permettent de stocker des chaînes à zéro terminal :

```
type
  TIdentificateur = array[0..15] of Char;
  TNomFic = array[0..259] of Char;
  TTexteMemo = array[0..1023] of WideChar;
```

La syntaxe étendue étant activée (**{$X+}**), vous pouvez affecter une constante chaîne à un tableau de caractères de base zéro. Les tableaux dynamiques ne peuvent pas être utilisés pour cela. Si vous initialisez une constante tableau avec une chaîne plus courte que la longueur déclarée du tableau, les caractères restants sont initialisés à #0. Pour davantage d'informations sur les tableaux, voir "Tableaux" à la page 5-19.

Utilisation des pointeurs, tableaux et des constantes chaînes

Pour manipuler des chaînes à zéro terminal, il est souvent nécessaire d'utiliser des pointeurs. Voir "Pointeurs et types pointeur" à la page 5-27.) Les constantes chaîne sont compatibles pour l'affectation avec les types *PChar et PWideChar* qui représentent des pointeurs sur des chaînes à zéro terminal de caractères *Char* ou *WideChar*. Par exemple :

```
var P: PChar;
  ⋮
P := 'Bonjour chez vous!';
```

fait pointer *P* sur une zone de la mémoire contenant une copie à zéro terminal de "Bonjour chez vous!", c'est équivalent à :

```
const TempChaine: array[0..19] of Char = 'Bonjour chez vous!'#0;
var P: PChar;
  ⋮
P := @TempChaine;
```

Vous pouvez également transmettre des constantes chaîne aux fonctions qui attendent des valeurs ou des paramètres **const** de type *PChar* ou *PWideChar*, par exemple `StrUpper('Bonjour chez vous!')`. Comme pour l'affectation d'un *PChar*, le compilateur génère une copie à zéro terminal de la chaîne et transmet à la fonction un pointeur sur cette copie. Vous pouvez, enfin, initialiser des constantes *PChar* ou *PWideChar* avec des littéraux chaîne, seuls ou dans un type structuré. Par exemple :

```
const
Message: PChar = 'Programme terminé';
  Invite: PChar = 'Entrez des valeurs: ';
  Chifres: array[0..9] of PChar = (
    'Zéro', 'Un', 'Deux', 'Trois', 'Quatre',
    'Cinq', 'Six', 'Sept', 'Huit', 'Neuf');
```

Les tableaux de caractères d'indice de base zéro sont compatibles avec *PChar* et *PWideChar*. Quand vous utilisez un tableau de caractères à la place d'une valeur pointeur, le compilateur convertit le tableau en une constante pointeur dont la valeur correspond à l'adresse du premier élément du tableau. Par exemple :

```
var
  MonTableau: array[0..32] of Char;
  MonPointeur: PChar;
begin
  MonTableau := 'Bonjour';
  MonPointeur := MyArray;
  UneProcedure(MonTableau);
  UneProcedure(MonPointeur);
end;
```

Ce code appelle deux fois *UneProcedure* avec la même valeur.

Un pointeur de caractère peut être indicé comme s'il était un tableau. Dans l'exemple précédent, `MonPointeur[0]` renvoie *B*. L'indice spécifie un décalage ajouté au pointeur avant de le déréferencer. Pour les variables *PWideChar*, l'indice est automatiquement multiplié par deux. Donc, si *P* est un pointeur de caractère, *P*`[0]` est équivalent à *P*`^` et spécifie le premier caractère du tableau, *P*`[1]` spécifie le second caractère du tableau, etc ; *P*`[-1]` spécifie le "caractère" immédiatement à gauche de *P*`[0]`. *Le compilateur n'effectue pas de vérification des bornes sur ces indices.*

La fonction *StrUpper* illustre l'utilisation d'indice de pointeur pour parcourir une chaîne à zéro terminal :

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;
```

Mélange de chaînes Pascal et de chaînes à zéro terminal

Vous pouvez mélanger des chaînes longues (des valeurs *AnsiString*) et des chaînes à zéro terminal (des valeurs *PChar*) dans les expressions ou les affectations ; vous pouvez aussi transmettre des valeurs *PChar* aux fonctions ou procédures qui attendent des paramètres de type chaîne longue. L'affectation *S* `:=` *P*, où *S* est une variable chaîne et *P* une expression *PChar*, copie une chaîne à zéro terminal dans une chaîne longue.

Dans une opération binaire, si un opérande est une chaîne longue et l'autre un *PChar*, l'opérande *PChar* est converti en chaîne longue.

Vous pouvez transtyper une valeur *PChar en chaîne longue*. Cela est pratique pour effectuer une opération de chaîne sur deux valeurs *PChar*. Par exemple :

```
opendir(PChar(Str));
```

(La déclaration d'*opendir* se trouve dans l'unité interface *Libc*.)

Vous pouvez également convertir une chaîne longue en une chaîne à zéro terminal. Les règles suivantes s'appliquent.

- Si *S* est une expression chaîne longue, `PChar(S)` convertit *S* en une chaîne à zéro terminal ; cela renvoie un pointeur sur le premier caractère de *S*.

  Avec Windows: Si, par exemple *Str1* et *Str2* sont ds chaînes longues, vous pouvez appeler la fonction *MessageBox* de l'API Win32 de la manière suivante :

  ```
  MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
  ```

  (*MessageBox* est déclarée dans l'unité interface *Windows*.)

  Avec Linux : Si, par exemple *Str* est une chaîne longue, vous pouvez appeler la fonction système *opendir* de la manière suivante :

  ```
  opendir(PChar(Str));
  ```

  (*opendir* est déclarée dans l'unité interface *Libc*.)

- Vous pouvez également utiliser `Pointer(S)` pour convertir une chaîne longue en un pointeur sans type. Mais si *S* est vide, le transtypage renvoie **nil**.
- Quand vous transtypez une variable chaîne longue en pointeur, le pointeur reste valide jusqu'à ce qu'une nouvelle valeur soit affectée à la variable ou que la variable sorte de la portée. Si vous transtypez une expression chaîne longue en pointeur, le pointeur n'est valable qu'à l'intérieur de l'instruction où le transtypage est effectué.
- Quand vous transtypez une expression chaîne longue en pointeur, le pointeur doit généralement être considéré comme étant en lecture seule. Vous pouvez utiliser sans risque le pointeur pour modifier la chaîne longue uniquement si les conditions suivantes sont respectées.
  - L'expression transtypée est une *variable chaîne longue*.
  - La chaîne n'est pas vide.
  - La chaîne est unique, c'est-à-dire que son compteur de référence vaut un. Pour vérifier que la chaîne est unique, appelez la procédure *SetLength*, *SetString* ou *UniqueString*.
  - La chaîne n'a pas été modifiée depuis le transtypage.
  - Les caractères modifiés se trouvent tous dans la chaîne. Faites attention à ne pas utiliser d'indice du pointeur hors intervalle.

Les mêmes règles s'appliquent aux mélanges de valeurs *WideString* à des valeurs *PWideChar*.

Types structurés

Les instances d'un type structuré contiennent plusieurs valeurs. Les types structurés sont les types *ensemble*, *tableau*, *enregistrement*, *fichier*, *classe*, *référence de classe et interface*. Pour des informations sur les classes et les références de classes, voir chapitre 7, "Classes et objets". Pour des informations sur les interfaces, voir chapitre 10, "Interfaces d'objets".) A l'exception des ensembles qui contiennent uniquement des valeurs scalaires, les types structurés peuvent contenir d'autres types structurés ; un type peut avoir un niveau illimité de structuration.

Par défaut, les valeurs d'un type structuré sont alignées sur des limites de mot ou de double-mot afin de disposer d'un accès plus rapide. Quand vous déclarez un type structuré, vous pouvez spécifier le mot réservé **packed** pour implémenter le stockage compressé des données. Par exemple :

```
type TNombress = packed array[1..100] of Real;
```

L'utilisation de **packed** ralentit l'accès aux données et, dans le cas d'un tableau de caractères, nuit à la compatibilité. Pour davantage d'informations, voir chapitre 11, "Gestion de la mémoire".

Ensembles

Un ensemble est une collection de valeurs ayant le même type scalaire. Les valeurs n'ont pas d'ordre intrinsèque, une même valeur ne peut donc pas apparaître deux fois dans un ensemble.

Les valeurs possibles du type ensemble sont tous les sous-ensembles du type de base, y compris l'ensemble vide. Le type de base ne peut avoir plus de 256 valeurs possibles et leur rang doit être compris entre 0 et 255. Toute construction de la forme :

```
set of typeBase
```

où *typeBase* est un type scalaire approprié, identifie un type ensemble.

En raison des limitations de taille des types de base, les types ensemble sont généralement définis avec des intervalles. Par exemple, les déclarations :

```
type
  TCertainEntiers = 1..250;
  TEnsembleEntiers = set of TCertainEntiers;
```

créent un type d'ensemble appelé *TEnsembleEntiers* dont les valeurs sont des collections d'entiers dans l'intervalle 1 à 250. Le même résultat est obtenu avec :

```
type TEnsembleEntiers = set of 1..250;
```

Etant donné cette déclaration, vous pouvez créer un ensemble de la manière suivante :

```
var Ens1, Ens2: TEnsembleEntiers;
 ⋮
Ens1 := [1, 3, 5, 7, 9];
Ens2 := [2, 4, 6, 8, 10]
```

Vous pouvez également utiliser directement la construction **set of ...** dans une déclaration de variables :

```
var MonEnsemble: set of 'a'..'z';
 ⋮
MonEnsemble := ['a','b','c'];
```

Voici d'autres exemples de type ensemble :

```
set of Byte
set of (Trefle, Carreau, Coeur, Pique)
set of Char;
```

L'opérateur **in** teste l'appartenance d'ensemble :

```
if 'a' in MonEnsemble then ... { faire quelque chose} ;
```

Chaque type d'ensemble peut contenir l'ensemble vide désigné par `[]`. Pour davantage d'informations sur les ensembles, voir "Constructeurs d'ensembles" à la page 4-15 et "Opérateurs d'ensembles" à la page 4-11.

Tableaux

Un tableau représente une collection indicée d'éléments de même type (appelé le *type de base*). Comme chaque élément a un indice unique, les tableaux (à la différences des ensembles) peuvent, sans ambiguïtés , contenir plusieurs fois la même valeur. Il est possible d'allouer des tableaux de manière *statique* ou *dynamique*.

Tableaux statiques

Les tableaux statiques sont désignés par des constructions de la forme :

array[*typeIndex*$_1$, ..., *typeIndex*$_n$] of *typeBase*

où chaque *typeIndex* est un type scalaire dont l'étendue de doit pas dépasser 2 Go. Comme les *typeIndex* indicent le tableau, le nombre d'éléments que le tableau peut contenir est limité par le produit de la taille des *typeIndex*. Pratiquement, *les typeIndex* sont en général des intervalles d'entiers.

Dans le cas le plus simple d'un tableau à une dimension, il n'y a qu'un seul *typeIndex*. Par exemple :

```
var MonTableau: array[1..100] of Char;
```

déclare une variable appelée *MonTableau* qui contient un tableau de 100 valeurs caractère. Etant donné cette déclaration, `MonTableau[3]` désigne le troisième caractère de *MonTableau*. Si vous créez un tableau statique sans affecter de valeurs à tous ses éléments, les éléments inutilisés sont néanmoins alloués et contiennent des données aléatoires ; ils sont identiques à des variables non initialisées.

Un tableau à plusieurs dimensions est un tableau de tableaux. Par exemple :

```
type TMatrice = array[1..10] of array[1..50] of Real;
```

est équivalent à :

```
type TMatrice = array[1..10, 1..50] of Real;
```

Quelle que soit la manière dont *TMatrice* est déclaré, ce type représente un tableau de 500 valeurs réelles. Une variable *MaMatrice de* type *TMatrice* peut être indicée comme ceci : `MaMatrice[2,45]` ou comme cela : `MaMatrice[2][45]`. De même :

```
packed array[Boolean,1..10,TPointures] of Integer;
```

est équivalent à :

```
packed array[Boolean] of packed array[1..10] of packed array[TPointures] of Integer;
```

Les fonctions standard *Low* et *High* acceptent les variables et les identificateurs de type tableau. Elles renvoient les bornes basse et haute du premier indice du type. La fonction standard *Length* renvoie le nombre d'éléments dans la première dimension du tableau.

Un tableau à une dimension compacté statique de valeurs *Char* est appelé une *chaîne compactée*. Les types chaîne compactée sont compatibles avec les types chaîne et avec les autres types de chaîne compactée ayant le même nombre d'éléments. Voir “Compatibilité et identité de types” à la page 5-37.

Un type tableau de la forme `array[0..x] of Char` est appelé un *tableau de caractères d'indice de base zéro*. Ces tableaux sont utilisés pour stocker des chaînes à zéro terminal et sont compatibles avec les valeurs *PChar*. Voir “Manipulation des chaînes à zéro terminal” à la page 5-14.

Tableaux dynamiques

Les tableaux dynamiques n'ont pas de taille ou de longueur fixe. La mémoire d'un tableau dynamique est réallouée quand vous affectez une valeur au tableau ou quand vous le transmettez à la procédure *SetLength*. Les types de tableau dynamique sont désignés par des constructions de la forme :

`array of` *typeBase*

Par exemple :

```
var MonTableauFlexible: array of Real;
```

déclare un tableau dynamique de réels à une dimension. La déclaration n'alloue pas de mémoire à *MonTableauFlexible*. Pour créer le tableau en mémoire, appelez *SetLength*. Par exemple, étant donné la déclaration précédente :

```
SetLength(MonTableauFlexible, 20);
```

alloue un tableau de 20 réels indicés de 0 à 19. Les tableaux dynamiques sont toujours indicés par des entiers en commençant toujours par 0.

Les variables tableau dynamique sont implicitement des pointeurs et sont gérés par la même technique de comptage de références que celle utilisée pour les chaînes longues. Pour libérer un tableau dynamique, affectez **nil** à une variable qui référence le tableau ou transmettez la variable à *Finalize* ; ces deux méthodes

libèrent le tableau dans la mesure où il n'y a pas d'autres références le désignant. Les tableaux dynamiques de longueur 0 ont la valeur **nil**. N'appliquez pas l'opérateur de déréférencement (^) à une variable tableau dynamique et ne la transmettez pas aux procédures *New et Dispose*.

Si *X* et *Y* sont des variables du même type de tableau dynamique, *X* := *Y* fait pointer *X* sur le même tableau que *Y*. (Il n'est pas nécessaire d'allouer de la mémoire à *X* avant d'effectuer cette opération.) A la différence des chaînes ou des tableaux statiques, les tableaux dynamiques ne sont pas copiés quand ils vont être modifiés. Donc, après l'exécution du code suivant :

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

La valeur de `A[0]` est 2. Si *A* et *B* étaient des tableaux statiques, `A[0]` vaudrait toujours 1.

L'affectation d'un indice d'un tableau dynamique (par exemple, `MonTableauFlexible[2] := 7`) ne réalloue pas le tableau. Les indices hors des bornes ne sont pas détectés à la compilation.

Quand des variables tableau dynamique sont comparées, les références sont comparées et non pas la valeur des tableaux. Donc, après l'exécution du code :

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
end;
```

`A = B` renvoie *False* mais `A[0] = B[0]` renvoie *True*.

Pour tronquer un tableau dynamique, transmettez-le à *SetLength* ou à *Copy* et réaffectez le résultat à la variable tableau. (La procédure *SetLength* est généralement plus rapide.) Si, par exemple, *A* est un tableau dynamique, `A := SetLength(A, 0, 20)` retranche tout sauf les vingt premiers éléments de *A*.

Une fois un tableau dynamique alloué, vous pouvez le transmettre aux fonctions standard *Length*, *High* et *Low*. *Length* renvoie le nombre d'éléments du tableau, *High* renvoie l'indice le plus élevé du tableau (c'est-à-dire *Length*–1) et *Low* renvoie 0. Dans le cas d'un tableau de longueur nulle, *High* renvoie –1 (avec cette anomalie que *High* < *Low*).

Remarque Dans les déclarations de certaines procédures et fonctions, les paramètres tableau sont représentés sous la forme `array of` *typeBase*, sans spécifier le type d'indice. Par exemple :

```
function VerifChaines(A: array of string): Boolean;
```

Cela indique que la fonction opère sur tous les tableaux du type de base spécifié indépendamment de leur taille, de leurs indices ou de leur allocation statique ou dynamique. Voir “Paramètres tableau ouvert” à la page 6-15.

Tableaux dynamiques multi-dimensionnels

Pour déclarer des tableaux dynamiques multi-dimensionnels, utilisez des constructions **array of ...** répétées. Par exemple :

```
type TGrilleMessage = array of array of string;
var Msgs: TGrilleMessage;
```

déclare un tableau de chaînes à deux dimensions. Pour instancier ce tableau, appelez *SetLength* avec deux arguments entiers. Donc, si *I* et *J* sont des variables de valeur entières :

```
SetLength(Msgs,I,J);
```

alloue un tableau de *I*-fois-*J* et `Msgs[0,0]` désigne un élément de ce tableau.

Vous pouvez créer des tableaux dynamiques multidimensionnels qui ne sont pas rectangulaires. Il faut tout d'abord appeler *SetLength* en lui transmettant les paramètres pour les n premières dimensions du tableau. Par exemple :

```
var Ints: array of array of Integer;
SetLength(Ints,10);
```

alloue dix lignes pour *Ints* mais pas les colonnes. Ultérieurement, vous pouvez allouer les colonnes une à une (en leur donnant des longueurs différentes) ; par exemple ;

```
SetLength(Ints[2], 5);
```

donne la longueur cinq à la troisième colonne de *Ints*. Arrivé là, même si aucune autre colonne n'a été allouée, vous pouvez affecter des valeurs à la troisième colonne ; par exemple, `Ints[2,4] := 6`.

L'exemple suivant utilise un tableau dynamique (et la fonction *IntToStr* déclarée dans l'unité *SysUtils*) pour créer une matrice triangulaire de chaînes.

```
var
  A : array of array of string;
  I, J : Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
  begin
    SetLength(A[I], I);
    for J := Low(A[I]) to High(A[I]) do
      A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
  end;
end;
```

Types tableau et affectations

Des tableaux sont compatibles pour l'affectation uniquement s'ils ont le même type. Comme Pascal utilise des équivalences de nom pour les types, le code suivant ne se compile pas :

```
var
  Int1: array[1..10] of Integer;
  Int2: array[1..10] of Integer;
  ⋮
Int1 := Int2;
```

Pour que l'affectation fonctionne, déclarez les variables comme suit :

```
var Int1, Int2: array[1..10] of Integer;
```

ou ainsi :

```
type TableauInt = array[1..10] of Integer;
var
  Int1: TableauInt;
  Int2: TableauInt;
```

Enregistrements

Un enregistrement (appelé aussi *structure* dans certains langages) représente un ensemble de données hétérogènes. Chaque élément est appelé un *champ* ; la déclaration d'un type enregistrement spécifie le nom et le type de chaque champ. Une déclaration de type enregistrement a la syntaxe suivante :

```
type nomTypeEnregistrement = record
  listeChamp1: type1;
  f
  listeChampn: typen;
end
```

où *nomTypeEnregistrement* est un identificateur valide et où chaque *type* désigne un type, et chaque *listeChamp* est un identificateur valide ou une liste d'identificateurs délimitée par des virgules. Le point-virgule final est facultatif.

Par exemple, la déclaration suivante crée un type enregistrement nommé *TDateRec*.

```
type
  TDateRec = record
    Annee: Integer;
    Mois: (Jan, Fev, Mar, Avr, Mai, Jun,
            Jul, Aou, Sep, Oct, Nov, Dec);
    Jour: 1..31;
  end;
```

Chaque *TDateRec* contient trois champs : une valeur entière appelée *Annee*, Une valeur d'un type énuméré appelé *Mois* et une autre valeur entière comprise entre 1 et 31 appelée *Jour*. Les identificateurs *Annee*, *Mois* et *Jour* sont des *noms de champs* de *TDateRec* qui se comportent comme des variables. Néanmoins, la déclaration de type *TDateRec* n'alloue pas de mémoire pour les champs *Annee*,

Mois et *Jour* ; la mémoire est allouée quand vous instanciez l'enregistrement, de la manière suivante :

```
var Record1, Record2: TDateRec;
```

Cette déclaration de variable crée deux instances de *TDateRec*, appelées *Record1* et *Record2*.

Vous pouvez accéder aux champs de l'enregistrement en qualifiant le nom de champ avec le nom de l'enregistrement :

```
Record1.Annee := 1904;
Record1.Mois := Jun;
Record1.Jour := 16;
```

Ou en utilisant une instruction **with** :

```
with Record1 do
begin
  Annee := 1904;
  Mois := Jun;
  Jour := 16;
end;
```

Vous pouvez alors copier les valeurs des champs de *Record1* dans *Record2* :

```
Record2 := Record1;
```

Comme la portée d'un nom de champ est limitée à l'enregistrement dans lequel il est spécifié, vous n'avez pas à vous préoccuper de conflits entre les noms de champ et des noms de variable.

Au lieu de définir des types d'enregistrement, vous pouvez utiliser directement la construction **record ...** dans des déclarations de variable :

```
var S: record
  Nom: string;
  Age: Integer;
end;
```

Cependant, une telle déclaration annule complètement les avantages des enregistrements, à savoir éviter le codage répétitif de groupes de variables identiques. De plus, des enregistrements déclarés séparément avec cette méthode ne sont pas compatibles pour l'affectation même si leur structure est identique.

Partie variable d'enregistrements

Un type enregistrement peut avoir une partie *variable* qui ressemble à une instruction **case**. La partie variable doit venir après les autres champs de la déclaration de l'enregistrement.

Pour déclarer un enregistrement avec une partie variable, utilisez la syntaxe suivante :

```
type nomTypeEnregistrement = record
  listeChamp1: type1;
  ƒ
  listeChampn: typen;
```

```
case sélecteur: typeScalaire of
  listeConstante1: (variante1);
 ƒ
  listeConstanten: (varianten);
end;
```

La première partie de la déclaration, jusqu'au mot réservé **case**, est identique à celle d'un type d'enregistrement standard. Le reste de la déclaration, du **case** jusqu'au point-virgule final facultatif, est appelé la partie variable. Dans la partie variable :

- *sélecteur* est facultatif, ce peut être tout identificateur valide. Si vous omettez *sélecteur*, omettez également le caractère deux points (**:**) après.
- *typeScalaire* désigne un type scalaire.
- Chaque *listeConstante* est une constante désignant une valeur du type *typeScalaire*, ou une liste de telles constantes délimitée par des virgules. Une valeur ne doit pas apparaître plus d'une fois dans tous les *listeConstante*.
- Chaque *variante* est une liste délimitée par des virgules de déclarations similaires aux constructions *listeChamp*: *type* de la partie principale du type enregistrement. Donc *variante* est de la forme :

  ```
  listeChamp1: type1;
    ⋮
  listeChampn: typen;
  ```

  où chaque *listeChamp* est un identificateur valide ou une liste d'identificateurs délimitée par des virgules, chaque *type* désigne un type et le point-virgule final est facultatif. Les *type*s ne peuvent pas être des chaînes longues, des tableaux dynamiques, des types *Variant*, des interfaces ou des types structurés contenant ces mêmes types. Par contre, ce peut être des pointeurs sur ces types.

Les enregistrements ayant une partie variable ont une syntaxe complexe mais une sémantique très simple. La partie variable d'un enregistrement contient plusieurs *variantes* qui partagent le même espace mémoire. Vous pouvez lire ou écrire dans tous les champs de toutes les variantes existantes à tout moment ; mais si vous écrivez dans un champ d'une des variantes puis dans un champ d'une autre variante, vous risquez d'écraser vos propres données. Le *sélecteur*, s'il est spécifié, fonctionne comme un champ supplémentaire (de type *typeScalaire*) dans la partie fixe de l'enregistrement.

Les parties variables ont deux rôles. Tout d'abord, vous pouvez avoir besoin de créer un enregistrement ayant des champs pour différentes sortes de données tout en sachant que vous n'utilisez jamais simultanément tous les champs dans une instance de l'enregistrement. Par exemple :

```
type
  TEmploye = record
  Prenom, Nom: string[40];
  Naissance: TDate;
  case Salarie: Boolean of
    True: (SalaireAnnuel: Currency);
```

```
    False: (TauxHoraire: Currency);
end;
```

L'idée ici est que chaque employé a soit un salaire soit un taux horaire, mais pas les deux. Donc, quand vous créez une instance de *TEmploye*, il n'y a pas de raison pour allouer de la mémoire pour les deux champs. Dans ce cas, la seule différence entre les *variant*es, c'est le nom du champ. Mais dans d'autres cas, les champs peuvent être de types différents. Voici des exemples plus compliqués :

```
type
  TPersonne = record
  Prenom, Nom: string[40];
  Naissance: TDate;
  case Citoyen: Boolean of
    True: (LieuNaissance: string[40]);
    False: (Pays: string[20];
            LieuEntree: string[20];
            DateEntree, DateSortie: TDate);
  end;

type
  TListeForme = (Rectangle, Triangle, Cercle, Ellipse, Autre);
  TFigure = record
    case TListeForme of
      Rectangle: (Hauteur, Largeur: Real);
      Triangle: (Cote1, Cote2, Angle: Real);
      Cercle: (Rayon: Real);
      Ellipse, Autre: ();
  end;
```

Pour chaque instance d'enregistrement, le compilateur alloue assez de mémoire pour contenir tous les champs de la variante la plus volumineuse. Le sélecteur optionnel et les *listeConstante* (comme *Rectangle*, *Triangle*, etc dans le dernier exemple) ne jouent aucun rôle dans la manière dont le compilateur gère les champs ; ils ne sont là que comme commodité pour le programmeur.

L'autre raison d'utiliser des parties variables, c'est qu'elles vous permettent de traiter les mêmes données comme si elles étaient de différents types, et ce même dans les cas où le compilateur n'autorise pas les transtypages. Si par exemple, vous avez un *Real* sur 64 bits comme premier champ dans une *variante* et un *Integer* sur 32 bits comme premier champ d'une autre variante. Vous pouvez alors affecter une valeur au champ *Real* puis en lire les 32 premiers bits comme valeur du champ *Integer* (et par exemple les transmettre à une fonction nécessitant un paramètre entier).

Types fichier

Un fichier est un ensemble ordonné d'éléments du même type. Les routines standard d'Entrées/Sorties utilisent les types prédéfinis *TextFile* et *Text* qui représentent un fichier contenant des caractères organisés en lignes. Pour davantage d'informations sur les entrées et sorties de fichier, voir chapitre 8, "Routines standard et Entrées/Sorties".

Pour déclarer un type fichier, utilisez la syntaxe :

```
type nomTypeFichier = file of type
```

où *nomTypeFichier* est un identificateur valide et *type* un type de taille fixe. Les types pointeur, implicites ou explicites ne sont pas permis. Un fichier ne peut donc pas contenir des tableaux dynamiques, des chaînes longues, des classes, des objets, des pointeurs, des variants, d'autres fichiers ou des types structurés en contenant.

Par exemple :

```
type
  EntreeRepertoire = record
    Prenom, Nom: string[20];
    Telephone: string[15];
    Liste: Boolean;
  end;
  Annuaire = file of EntreeRepertoire;
```

déclare un type fichier pour enregistrer des noms et des numéros de téléphone.

Vous pouvez également utiliser directement la construction **file of ...** dans une déclaration de variable. Par exemple :

```
var List1: file of EntreeRepertoire;
```

Le mot **file** seul indique un fichier sans type :

```
var DataFile: file;
```

Pour davantage d'informations, voir “Fichiers sans type” à la page 8-4.

Les fichiers ne sont pas autorisés dans les tableaux ou les enregistrements.

Pointeurs et types pointeur

Un pointeur est une variable qui désigne une adresse mémoire. Quand un pointeur contient l'adresse d'une autre variable, on dit qu'il *pointe* sur l'emplacement en mémoire de cette variable ou sur les données qui y sont stockées. Dans le cas d'un tableau ou d'un type structuré, un pointeur contient l'adresse du premier élément de la structure.

Les pointeurs sont *typés* afin d'indiquer le type de données stockées à l'adresse qu'ils contiennent. Le type général *Pointer* peut représenter un pointeur sur tous les types de données alors que d'autres pointeurs, plus spécialisés, ne pointent que sur des types de données spécifiques. Les pointeurs occupent quatre octets en mémoire.

Présentation des pointeurs

Pour comprendre comment les pointeurs fonctionnent, examinez l'exemple suivant :

```
1   var
2     X, Y: Integer;   // X et Y sont des variables Integer
3     P: ^Integer;     // P pointe sur un Integer
4   begin
5     X := 17;         // affecte une valeur à X
6     P := @X;         // affecte l'adresse de X à P
7     Y := P^;         // déréférence P; affecte le résultat à Y
8   end;
```

La ligne 2 déclare *X* et *Y* comme variables de type *Integer*. La ligne 3 déclare *P* comme un pointeur sur une valeur *Integer* ; cela signifie que *P* peut pointer sur l'adresse de *X* ou de *Y*. La ligne 5 affecte une valeur à *X* et la ligne 6 affecte l'adresse de *X* (désignée par `@X`) à *P*. Enfin, la ligne 7 récupère la valeur située à l'emplacement pointé par *P* (désigné par `^P`) et l'affecte à *Y*. Après l'exécution de ce code, *X* et *Y* ont la même valeur, à savoir 17.

L'opérateur @ utilisé ici pour obtenir l'adresse d'une variable agit également sur les fonctions et les procédures. Pour davantage d'informations, voir "L'opérateur @" à la page 4-13 et "Types procédure dans les instructions et les expressions" à la page 5-32.

Le symbole **^** a deux fonctions, toutes deux illustrées dans l'exemple précédent. Quand il apparaît avant un identificateur de type :

^*nomType*

il désigne un type qui représente des pointeurs sur des variables de type *nomType*. Quand il apparaît après une variable pointeur :

pointeur^

il *déréférence* le pointeur, c'est-à-dire qu'il renvoie la valeur stockée à l'adresse mémoire contenue dans le pointeur.

Cet exemple peut sembler un moyen bien compliqué pour copier la valeur d'une variable dans une autre, puisque cela peut s'effectuer par une simple instruction d'affectation. Mais les pointeurs sont utiles pour plusieurs raisons. Tout d'abord, comprendre les pointeurs permet de mieux comprendre le Pascal Objet, car fréquemment des pointeurs agissent en sous-main dans le code, même quand ils n'apparaissent pas explicitement. Tout type de données nécessitant des blocs de mémoire importants alloués dynamiquement utilise des pointeurs. Ainsi, les variables chaîne longue sont implicitement des pointeurs, tous comme les variables classe. De plus, certaines techniques de programmation avancée nécessitent l'utilisation de pointeurs.

Enfin, les pointeurs sont parfois le seul moyen de contourner les exigences du Pascal Objet sur le typage des données. En faisant référence à une variable à l'aide d'un *Pointer* générique, en transtypant ce *Pointer* dans un type plus spécifique, puis en le déréférençant, vous pouvez traiter les données stockées dans n'importe quelle variable comme appartenant à un type quelconque. Par

exemple, le code suivant affecte les données stockées dans une variable réelle à une variable entière.

```
type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  ⋮
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

Bien évidemment, les réels et les entiers ne sont pas stockés en utilisant le même format. Cette affectation copie simplement des données binaires brutes de *R* vers *I* sans les convertir.

Outre l'affectation du résultat d'une opération @, plusieurs routines standard permettent d'affecter une valeur à un pointeur. Les procédures *New* et *GetMem* affectent une adresse mémoire à un pointeur existant, alors que les fonctions *Addr* et *Ptr* renvoient un pointeur sur l'adresse ou la variable spécifiée.

Il est possible de qualifier un pointeur déréférencé ou de l'utiliser comme qualificateur, comme dans l'expression `P1^.Data^`.

Le mot réservé **nil** est une constante spéciale qui peut être affectée à tout pointeur. Quand la valeur **nil** est affectée à un pointeur, le pointeur ne désigne plus rien.

Types pointeur

Il est possible de déclarer tout type de pointeur en utilisant la syntaxe :

`type` *nomTypePointeur* `= ^`*type*

Quand vous définissez un enregistrement ou d'autres types de données, il est courant de définir un pointeur sur ce type. Cela simplifie la manipulation des instances de ce type sans avoir à copier de gros blocs de mémoire.

Les types standard de pointeur ont de nombreuses fonctions. Le type le plus versatile, *Pointer* peut pointer sur les données de tout type. Mais une variable *Pointer* ne peut être déréférencée : placer le symbole **^** après une variable *Pointer* déclenche une erreur de compilation. Pour accéder aux données référencées par une variable *Pointer*, il faut d'abord la transtyper dans un autre type de pointeur puis alors seulement la déréférencer.

Pointeurs de caractère

Les types fondamentaux *PAnsiChar* et *PWideChar* représentent des pointeurs sur, respectivement, des valeurs *AnsiChar* et *WideChar*. Le type générique *PChar* représente un pointeur sur un *Char* (dans l'implémentation en cours, c'est un *AnsiChar*. Ces pointeurs de caractère sont utilisés pour manipuler des chaînes à zéro terminal. Pour davantage d'informations, voir “Manipulation des chaînes à zéro terminal” à la page 5-14.)

Autres types standard de pointeurs

Les unités *System* et *SysUtils* déclarent plusieurs types de pointeur standard couramment utilisés.

Tableau 5.6 Types de pointeur déclarés dans les unités System SysUtils

Type de pointeur	Pointe sur des variables de type
PAnsiString, *PString*	*AnsiString*
PByteArray	*ByteArray* (déclaré dans *SysUtils*). Utilisé pour transtyper dynamiquement de la mémoire allouée pour les tableaux.
PCurrency, *PDouble*, *PExtended*, *PSingle*	*Currency*, *Double*, *Extended*, *Single*
PCurrency, *PDouble*, *PExtended*, *PSingle*	*Currency*, *Double*, *Extended*, *Single*
PInteger	*Integer*
POleVariant	*OleVariant*
PShortString	*ShortString*. Utilisé pour adapter du code ancien utilisant le type *PString*.
PTextBuf	*TextBuf* (déclaré dans *SysUtils*). *TextBuf* est le type interne de tampon d'un enregistrement fichier *TTextRec*.
PVarRec	*TVarRec* (déclaré dans *System*)
PVariant	*Variant*
PWideString	WideString
PWordArray	*TWordArray* (déclaré dans *SysUtils*). Utilisé pour transtyper dynamiquement de la mémoire allouée pour des tableaux de valeurs sur deux octets.

Types procédure

Les types procédure permettent de traiter des procédures et des fonctions comme des valeurs pouvant être affectées à des variables ou transmises à d'autres procédures ou fonctions. Si, par exemple, vous définissez une fonction appelée *Calc* qui prend deux paramètres entiers et renvoie un entier :

```
function Calc(X,Y: Integer): Integer;
```

Vous pouvez affecter la fonction *Calc* à la variable *F* :

```
var F: function(X,Y: Integer): Integer;
F := Calc;
```

Si vous prenez un en-tête de fonction ou de procédure et supprimez l'identificateur suivant le mot **procedure** ou **function**, ce qui reste est la définition d'un type procédure. Vous pouvez utiliser directement cette déclaration de type dans une déclaration de variable (comme dans l'exemple précédent) ou déclarer de nouveaux types :

```
type
  TFonctionEntiere = function: Integer;
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFonc = function(X: Double): Double;
var
  F: TFonctionEntiere;          { F est une fonction sans paramètre qui renvoie un entier }
  Proc: TProcedure;             { Proc est une procédure sans paramètre }
  SP: TStrProc;                 { SP est une procédure attendant un paramètre chaîne }
  M: TMathFonc;                 { M est une fonction qui prend un paramètre Double (réel)
                                    et renvoie un Double }
procedure FuncProc(P: TFonctionEntiere); {FuncProc est une procédure dont le seul paramètre
                                           est une fonction entière sans paramètre}
```

Les variables ci-dessus sont toutes des *pointeurs de procédure, c'est-à-dire des pointeurs sur l'adresse d'une procédure ou d'une fonction*. Pour pointer la méthode d'une instance d'objet (voir chapitre 7, “Classes et objets”), vous devez ajouter les mots **of object** au nom de type. Par exemple :

```
type
  TMethode = procedure of object;
  TNotifyEvent = procedure(Sender: TObject) of object;
```

Ces types sont des *pointeurs de méthode*. Un pointeur de méthode est en fait une paire de pointeurs, le premier stocke l'adresse d'une méthode et le second une référence à l'objet auquel appartient la méthode. Etant donné les déclarations :

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ⋮
  end;
var
  MainForm: TMainForm;
  OnClick: TNotifyEvent
```

vous pouvez utiliser l'affectation suivante :

```
OnClick := MainForm.ButtonClick;
```

Deux types de procédure sont compatibles si :

- Ils ont la même convention d'appel.
- Il renvoient le même type de valeur ou pas de valeur.
- Ils ont le même nombre de paramètres, avec le même type aux mêmes positions. Le nom des paramètres est sans importance.

Les types de pointeur de procédure sont toujours incompatibles avec les types de pointeur de méthode. La valeur **nil** peut être affectée à tous les types de procédure.

Les procédures et fonctions imbriquées (c'est-à-dire les routines déclarées à l'intérieur d'autres routines) ne peuvent s'utiliser comme valeur procédurale (de type procédure) tout comme les fonctions et procédure prédéfinies. Si vous voulez utiliser une routine prédéfinie comme *Length* en tant que valeur procédurale, écrivez une routine qui l'encapsule :

```
function FLength(S: string): Integer;
begin
  Result := Length(S);
end;
```

Types procédure dans les instructions et les expressions

Quand une variable procédurale se trouve dans la partie gauche d'une instruction d'affectation, le compilateur attend également une valeur procédurale à droite. L'affectation fait de la variable placée à gauche un pointeur sur la fonction ou la procédure indiquée à droite de l'affectation. Néanmoins, dans d'autres contextes, l'utilisation d'une variable procédurale produit un appel de la procédure ou de la fonction référencée. Vous pouvez même utiliser une variable procédurale pour transmettre des paramètres :

```
var
  F: function(X: Integer): Integer;
  I: Integer;
function UneFonction(X: Integer): Integer;
 ⋮
F := SomeFunction;  // affecte SomeFunction à F
I := F(4);          // appel de la fonction et affectation du résultat à I
```

Dans les instructions d'affectation, le type de la variable à gauche détermine l'interprétation des pointeurs de procédure ou de méthode à droite. Par exemple :

```
var
  F, G: function: Integer;
  I: Integer;
function UneFonction: Integer;
 ⋮
F := UneFonction;   // affecte UneFonction à F
G := F;             // copie F dans G
I := G;             // appelle la fonction, affecte le résultat à I
```

La première instruction affecte une valeur procédurale à *F*. La deuxième instruction copie cette valeur dans une autre variable. La troisième instruction appelle la fonction référencée et affecte le résultat à *I*. Comme *I* est une variable entière et pas une valeur procédurale, la dernière instruction appelle réellement la fonction (qui renvoie un entier).

Dans certaines situations, l'interprétation d'une variable procédurale n'est pas toujours aussi évidente. Soit l'instruction :

```
if F = MaFonction then ...;
```

Ici, l'occurrence de *F* produit un appel de fonction : le compilateur appelle la fonction pointée par *F* puis la fonction *MaFonction* et compare les résultats. La règle veut qu'à chaque fois qu'une variable procédurale apparaît dans une expression, elle représente un appel à la procédure ou la fonction référencée. Dans le cas où *F* référence une procédure, qui ne renvoie pas de valeur, ou si *F* désigne une fonction nécessitant des paramètres, l'instruction précédente déclenche une erreur de compilation. Pour comparer la valeur procédurale de *F* à *MaFonction*, utilisez :

```
if @F = @MaFonction then ...;
```

`@F` convertit *F* en une variable pointeur sans type contenant une adresse et `@MaFonction` renvoie l'adresse de *MaFonction*.

Pour obtenir l'adresse mémoire d'une variable procédurale et pas l'adresse qu'elle contient, utilisez @@. Ainsi, `@@F` renvoie l'adresse de *F*.

L'opérateur @ peut également être utilisé pour affecter une valeur de pointeur sans type à une variable procédurale. Par exemple ;

```
var StrComp: function(Str1, Str2: PChar): Integer;
 ⋮
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

appelle la fonction *GetProcAddress* et fait pointer *StrComp* sur le résultat.

Toute variable procédurale peut contenir la valeur **nil,** ce qui signifie qu'elle ne pointe sur rien. Mais essayer *d'appeler* **une** variable procédurale de valeur **nil** est une erreur. Pour tester si une variable procédurale est initialisée, utilisez la fonction standard *Assigned* :

```
if Assigned(OnClick) then OnClick(X);
```

Types variants

Il est parfois nécessaire de manipuler des données dont le type change ou est inconnu lors de la compilation. Dans ce cas, une des solutions consiste à utiliser des variables et des paramètres de type *Variant* qui représentent des valeurs dont le type peut changer à l'exécution. Les variants offrent une plus grande flexibilité que les variables standard mais consomment davantage de mémoire. De plus les opérations où les variants sont utilisés sont plus lentes que celles portant sur des types associés statiquement. Enfin, les opérations illégales portant sur des variants provoquent fréquemment des erreurs d'exécution, alors que les mêmes erreurs avec des variables normales sont détectées à la compilation. Vous pouvez également créer des types de variants personnalisés.

Les variants peuvent contenir des valeurs de tout type à l'exception des enregistrements, des ensembles, des tableaux statiques, des classes, des références de classe et des pointeurs. En d'autres termes, les variants peuvent tout contenir

sauf les types structurés et les pointeurs. Ils peuvent contenir des objets COM et CORBA, dont les méthodes et les propriétés peuvent être accédées grâce à eux. (Voir chapitre 10, “Interfaces d’objets”.) Ils peuvent contenir des tableaux dynamiques et une forme particulière de tableaux statiques appelée *tableau variant*. Pour davantage d’informations, voir “Tableaux variants” à la page 5-36.) Il est possible de mélanger dans les expressions et les affectations des variants à des valeurs entières, réelles, chaînes ou booléennes : le compilateur effectue automatiquement les conversions de type.

Il n’est pas possible d’indicer les variants contenant des chaînes. Donc, si *V* est un variant contenant une valeur chaîne, la construction `V[1]` provoqie une erreur d’exécution.

Un variant occupe 16 octets de mémoire, il est constitué d’un code de type et d’une valeur ou d’un pointeur sur une valeur ayant le type spécifié par le code. Tous les variants sont initialisés à la création avec une valeur spéciale *Unassigned*. La valeur spéciale *Null* indique des données inconnues ou manquantes.

La fonction standard *VarType* renvoie le code de type d’un variant. La constante *varTypeMask* est un masque de bits utilisé pour extraire le code de la valeur renvoyée par *VarType*. Par exemple :

```
VarType(V) and varTypeMask = varDouble
```

renvoie *True* si *V* contient une valeur *Double* ou un tableau de *Double*. Le masque cache simplement le premier bit qui indique si le variant contient un tableau. Le type d’enregistrement *TVarData* défini dans l’unité *System* peut être utilisé pour transtyper des variants et accéder à leur représentation interne. Pour la liste des codes renvoyés par *VarType*, voir l’aide en ligne. De nouveaux codes seront éventuellement ajoutés dans les versions ultérieures du Pascal Objet.

Conversions de types variants

Tous les types entiers, réels, chaîne, caractère et booléens sont compatibles pour l’affectation avec le type *Variant*. Les expressions peuvent être explicitement transtypées en variants. Les fonctions standard *VarAsType* et *VarCast* permettent de modifier la représentation interne d’un variant. Le code suivant illustre l’utilisation de variants et certaines conversions effectuées automatiquement quand des variants sont combinés avec d’autres types.

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
begin
  V1 := 1; { valeur entière}
  V2 := 1234.5678; { valeur réelle }
  V3 := 'Bonjour chez vous!'; { valeur chaîne }
  V4 := '1000'; { valeur chaîne }
  V5 := V1 + V2 + V4; { valeur réelle 2235.5678}
  I := V1; { I = 1 (valeur entière) }
```

```
    D := V2; { D = 1234.5678 (valeur réelle) }
    S := V3; { S = 'Bonjour chez vous!' (valeur chaîne) }
    I := V4; { I = 1000 (valeur entière) }
    S := V5; { S = '2235.5678' (valeur chaîne) }
  end;
```

Le compilateur effectue les conversions de type en respectant les règles suivantes.

Tableau 5.7 Règles de conversion des types variants

Cible Source	**entier**	**réel**	**chaîne**	**caractère**	**booléen**
entier	convertit au format entier	convertit en réel	convertit en représentation chaîne	pareil que pour une chaîne (à gauche)	renvoie *False* si la valeur est 0 et *True* sinon
réel	arrondit à l'entier le plus proche	convertit au format réel	convertit en chaîne en utilisant les paramètres régionaux	pareil que pour une chaîne (à gauche)	renvoie *False* si la valeur est 0 et *True* sinon
chaîne	convertit en entier en tronquant si nécessaire. Déclenche une exception si la chaîne n'est pas numérique.	convertit en réel en utilisant les paramètres régionaux. Déclenche une exception si la chaîne n'est pas numérique.	convertit au format chaîne/ caractère	pareil que pour une chaîne (à gauche)	renvoie *False* si la chaîne est "false" (pas de différence majuscule/ minuscule) ou une chaîne numérique qui s'évalue à 0, *True* si la chaîne est "true" ou une chaîne numérique non nulle ; sinon déclenche une exception
caractère	pareil que pour une chaîne (ci-dessus)	pareil que pour une chaîne (ci-dessus)	pareil que pour une chaîne (ci-dessus)	pareil que pour une chaîne	pareil que pour une chaîne (ci-dessus)
booléen	*False* = 0, *True* = –1 (255 si *Byte*)	*False* = 0, *True* = –1	*False* = "0", *True* = "–1"	pareil que pour une chaîne (à gauche)	*False* = *False*, *True* = *True*
Unassigned	renvoie 0	renvoie 0	renvoie une chaîne vide	pareil que pour une chaîne (à gauche)	renvoie *False*
Null	déclenche une exception	déclenche une exception	déclenche une exception	pareil que pour une chaîne (à gauche)	déclenche une exception

Les affectations hors de l'étendue provoquent fréquemment l'affectation à la variable cible de la plus grande valeur possible de son étendue. Les affectations illégales déclenchent une exception *EVariantError*.

Des règles de conversion particulières s'appliquent au type réel *TDateTime* déclaré dans l'unité *System*. Quand un *TDateTime* est converti dans tout autre type, il est traité comme une valeur *Double normale*. Quand un entier, un réel ou un booléen est converti en un *TDateTime*, il est tout d'abord converti en *Double* puis lu comme une valeur date-heure. Quand une chaîne est convertie en un *TDateTime*, elle est interprétée comme une valeur date-heure en utilisant les paramètres régionaux. Quand une valeur *Unassigned* est convertie en *TDateTime*, elle est traitée comme une valeur réelle ou entière nulle. La conversion de la valeur *Null* en *TDateTime* déclenche une exception.

Sous Windows, si un variant référence un objet COM, toute tentative de conversion lit la propriété par défaut l'objet et convertit la valeur au type requis. Si l'objet n'a pas de propriété par défaut, une exception est déclenchée.

Utilisation de variants dans les expressions

Tous les opérateurs, sauf **^**, **is** et **in** acceptent des opérandes variants. Les opérations sur les variants renvoient des valeurs de type *Variant* ; elles renvoient *Null* si un ou les deux opérandes valent *Null* et déclenchent une exception quand l'un ou les deux opérandes ont la valeur *Unassigned*. Dans une opération binaire, si un seul des opérandes est un variant, l'autre est converti en variant.

Le type du résultat d'une opération est déterminé par ses opérandes. En général, les règles qui s'appliquent aux variants sont les mêmes que pour les opérandes de types définis statiquement. Par exemple, si *V1* et *V2* sont des variants contenant une valeur entière et une valeur réelle, `V1 + V2` renvoient un variant de valeur réelle. Pour davantage d'informations sur le résultat d'une opération, voir "Opérateurs" à la page 4-6. Cependant, il est possible d'effectuer sur des combinaisons de valeurs variants des opérations qui ne sont pas autorisées avec des expressions dont le type est déterminé statiquement. Dans la mesure du possible, le compilateur convertit les variants inadaptés en utilisant les règles indiquées dans le tableau 5.7. Par exemple, si *V3* et *V4* sont des variants contenant une chaîne numérique et un entier, l'expression `V3 + V4` renvoie un variant de valeur entière : la chaîne numérique est convertie en entier avant d'effectuer l'opération.

Tableaux variants

Il n'est pas possible d'affecter un tableau statique ordinaire à un variant. Vous pouvez, à la place, créer un *tableau variant* en appelant l'une des fonctions standard *VarArrayCreate* ou *VarArrayOf*. Par exemple :

```
V: Variant;
 ⋮
V := VarArrayCreate([0,9], varInteger);
```

crée un tableau variant d'entiers de longueur 10 et l'affecte au variant *V*. Il est possible d'indicer le tableau en utilisant `V[0]`, `V[1]`, etc. Mais il n'est pas possible de transmettre un élément de tableau variant comme paramètre **var**. Les tableaux variants sont toujours indicés par des entiers.

Le deuxième paramètre dans l'appel de *VarArrayCreate* est le code du type de base du tableau. Pour une liste de ces codes, voir l'aide en ligne sur *VarType*. Ne transmettez jamais le code *varString* à *VarArrayCreate* ; pour créer un tableau variant de chaînes, utilisez *varOleStr*.

Les variants peuvent contenir des tableaux variants de différentes tailles, dimensions et types de base. Les éléments d'un tableau variant peuvent être de n'importe quel type admis dans les variants sauf *ShortString* et *AnsiString. De plus,* si le type de base du tableau est *Variant*, ses éléments peuvent même être hétérogènes. Utilisez la fonction *VarArrayRedim* pour redimensionner un tableau variant. Les autres fonctions standard agissant sur les tableaux variants sont *VarArrayDimCount*, *VarArrayLowBound*, *VarArrayHighBound*, *VarArrayRef*, *VarArrayLock* et *VarArrayUnlock*.

Quand un variant contenant un tableau variant est affecté à un autre variant ou transmis comme paramètre par valeur, la totalité du tableau est copiée. Vous ne devez donc effectuer de telles opérations que si elles sont réellement nécessaires vu leur coût en mémoire.

OleVariant

Le type *OleVariant* existe sur les deux plates-formesWindows et Linux. La différence principale entre *Variant* et *OleVariant* est que *Variant* peut contenir des types de données que seule l'application en cours sait traiter. *OleVariant* contient uniquement des types de données compatibles avec Ole Automation, ce qui signifie que ces types de données peuvent être transférés entre programmes ou sur le réseau sans qu'il soit nécessaire de savoir si l'autre extrémité saura manipuler les données.

Quand vous affectez un *Variant* qui contient des données personnalisées (comme une chaîne Pascal ou un des nouveaux types variants personnalisés) à un *OleVariant*, la bibliothèque d'exécution essaie de convertir le *Variant* en l'un des types de données *OleVariant* standard (une chaîne Pascal est convertie en chaîne Ole BSTR). Par exemple, si un variant contenant une chaîne *AnsiString* est affecté à un *OleVariant*, la chaîne *AnsiString* devient une *WideString*. La même chose est vraie lorsque vous passez un *Variant* au paramètre *OleVariant* d'une fonction.

Compatibilité et identité de types

Pour comprendre quelles opérations peuvent être effectuées sur quelles expressions, il est nécessaire de distinguer diverses formes de compatibilité dans les types et les valeurs. Entre autre, il faut différencier *identité de type*, *compatibilité de type* et *compatibilité pour l'affectation*.

Identité de types

L'identité de types est assez évidente. Quand un identificateur de type est déclaré en utilisant, sans qualification, un autre identificateur de type, les deux désignent le même type. Ainsi, étant donné les déclarations suivantes :

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

T1, *T2*, *T3*, *T4* et *Integer* désignent tous le même type. Pour créer des types distincts, répétez le mot **type** dans la déclaration. Par exemple :

```
type TMonEntier = type Integer;
```

crée un nouveau type appelé *TMonEntier* différent du type *Integer*.

Les constructions du langage qui fonctionnent comme des noms de type désignent un type différent à chaque nouvelle occurrence. Ainsi, les déclarations suivantes :

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

créent deux types distincts, *TS1* et *TS2*. De même les déclarations de variable :

```
var
  S1: string[10];
  S2: string[10];
```

créent deux variables de types distincts. Pour créer des variables du même type, utilisez :

```
var S1, S2: string[10];
```

ou

```
type MaChaine = string[10];
var
  S1: MaChaine;
  S2: MaChaine;
```

Compatibilité de types

Chaque type est compatible avec lui-même. Deux types distincts sont compatibles s'ils vérifient au moins une des conditions suivantes :

- Ces sont tous les deux des types réels.
- Ce sont tous les deux des types entiers.
- Un type est un intervalle de l'autre.
- Les deux types sont des intervalles du même type.

- Les deux sont des types ensemble dont les types de base sont compatibles.
- Les deux sont des types chaîne compactée ayant le même nombre de composants.
- L'un est un type chaîne et l'autre est de type chaîne, chaîne compactée ou *Char*.
- L'un est un type *Variant* et l'autre est de type entier, réel, chaîne, caractère ou booléen.
- Les deux sont des type classe, référence de classe ou interface et un des types est dérivé de l'autre.
- L'un des types est *PChar* ou *PWideChar* et l'autre est un tableau de caractères d'indice de base zéro de la forme `array[0..n] of Char`.
- L'un des types est *Pointer* (un pointeur sans type) et l'autre est un type quelconque de pointeur.
- Les deux types sont des pointeurs sur le même type et la directive de compilation **{$T+}** est activée.
- Les deux sont des types procédure avec le même type de résultat, le même nombre de paramètres, et le même type de paramètre à la même position.

Compatibilité pour l'affectation

La compatibilité pour l'affectation n'est pas une relation symétrique. Une expression de type *T2* peut être affectée à une variable de type *T1* si la valeur de l'expression est dans l'étendue de *T1* et si au moins une des conditions suivantes est vérifiée :

- *T1* et *T2* sont du même type, et ce n'est pas un type fichier ou un type structuré contenant un type fichier à un niveau quelconque.
- *T1* et *T2* sont des types scalaires compatibles.
- *T1* et *T2* sont tous les deux des types réels.
- *T1* est un type réel et *T2* un type entier.
- *T1* est de type *PChar* ou d'un type chaîne quelconque et l'expression est une constante chaîne.
- *T1* et *T2* sont tous les deux des types chaîne.
- *T1* est un type chaîne et *T2* est un *Char* ou un type chaîne compactée.
- *T1* est une chaîne longue et *T2* est un *PChar*.
- *T1* et *T2* sont des types de chaîne compactée compatibles.
- *T1* et *T2* sont des types ensemble compatibles.
- *T1* et *T2* sont des types pointeur compatibles.

- *T1* et *T2* sont tous les deux des types classe, référence de classe ou interface et *T2* est dérivé de *T1*.
- *T1* est un type interface et *T2* est un type de classe qui implémente *T1*.
- *T1* est un *PChar* ou un *PWideChar* et *T2* est un tableau de caractères d'indice de base zéro de la forme `array[0..n] of Char`.
- *T1* et *T2* sont des type procédure compatibles. Un identificateur de fonction ou de procédure est traité dans certaines instructions d'affectation comme une expression de type procédure. Voir “Types procédure dans les instructions et les expressions” à la page 5-32.
- *T1* est un *Variant* et *T2* est un type entier, réel, chaîne, caractère, booléen ou interface.
- *T1* est un type entier, réel, chaîne, caractère ou booléen et *T2* est *Variant*.
- *T1* est le type interface *IUnknown* ou *IDispatch* et *T2* est un *Variant*. Le code de type du variant doit être *varEmpty*, *varUnknown* ou *varDispatch* si *T1* est *IUnknown* et *varEmpty* ou *varDispatch* si *T1* est *IDispatch*.

Déclaration de types

Une déclaration de type spécifie un identificateur désignant un type. La syntaxe d'une déclaration de type est :

```
type nouveauNomType = type
```

où *nouveauNomType* est un identificateur valide. Par exemple, étant donné la déclaration de type suivante :

```
type TMaChaine = string;
```

vous pouvez faire la déclaration de variable suivante :

```
var S: TMaChaine;
```

La portée d'un identificateur de type n'inclut pas la déclaration de type même (sauf pour les types pointeur). Il n'est donc pas possible de définir, par exemple, un type enregistrement qui s'utilise lui-même de manière récursive.

Quand vous déclarez un type identique à un type existant, le compilateur traite le nouvel identificateur de type comme un alias de l'ancien. Ainsi, étant donné les déclarations suivantes :

```
type TValeur = Real;
var
  X: Real;
  Y: TValeur;
```

X et *Y* sont du même type : à l'exécution, il n'est pas possible de distinguer *TValeur* de *Real*. Généralement, cela ne pose pas problème, mais si vous définissez un nouveau type pour utiliser des informations de type à l'exécution (par exemple, pour associer un éditeur de propriété aux propriétés d'un type

donné), la distinction entre “nom différent” et “type différent” devient importante. Dans un tel cas, utilisez la syntaxe :

```
type nouveauNomType = type type
```

Par exemple :

```
type TVlaleur = type Real;
```

oblige le compilateur à créer un nouveau type distinct appelé *TValeur*.

Variables

Une variable est un identificateur dont la valeur peut être modifiée à l’exécution. Exprimée différemment, une variable est le nom d’un emplacement mémoire ; le nom vous permet de lire ou d’écrire l’emplacement mémoire. Les variables sont comme des conteneurs de données qui, comme elles sont typées, peuvent indiquer au compilateur comment interpréter les données qu’elles contiennent.

Déclaration de variables

La syntaxe de base de la déclaration d’une variable est :

```
var listeIdentificateur: type;
```

où *listeIdentificateur* est une liste d’identificateurs valides délimitée par des virgules et *type* un type valide. Par exemple :

```
var I: Integer;
```

déclare une variable *I* de type *Integer*, et :

```
var X, Y: Real;
```

déclare deux variables, *X* et *Y*, de type *Real*.

Dans des déclarations de variables consécutives, il n’est pas nécessaire de répéter le mot réservé **var** :

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Chiffres: 0..9;
  Okay: Boolean;
```

Les variables déclarées dans une procédure ou une fonction sont parfois appelées *locales* alors que les autres variables sont appelées *globales*. Les variables globales peuvent être initialisées lors de leur création en utilisant la syntaxe :

```
var identificateur: type = expressionConstante;
```

où *expressionConstante* est une expression constante représentant une valeur de type *type*. Pour davantage d’informations sur les expressions constantes, voir “Expressions constantes” à la page 5-45.) Ainsi, la déclaration :

```
var I: Integer = 7;
```

est équivalente au code suivant :

```
var I: Integer;
 ⋮
I := 7;
```

Une déclaration de plusieurs variables (de la forme `var X, Y, Z: Real;`) ne peut comporter d'initialisation, ni ne peut déclarer des variables de type fichier et variant.

Si vous n'initialisez pas explicitement une variable globale, le compilateur l'initialise à 0. Les variables locales, par contre, ne peuvent être initialisées dans leur déclaration et contiennent des données aléatoires tant qu'elles ne sont pas initialisées.

Quand vous déclarez une variable, vous allouez de la mémoire qui est libérée automatiquement quand la variable n'est plus utilisée. En particulier, les variables locales n'existent que jusqu'à ce que le programme sorte de la fonction ou de la procédure qui les a déclarées. Pour davantage d'informations sur les variables et la gestion mémoire, voir chapitre 11, “Gestion de la mémoire”.

Adresses absolues

Vous pouvez créer une nouvelle variable placée à la même adresse qu'une variable existante. Pour cela, placez le mot **absolute** après le nom de type, dans la déclaration de la nouvelle variable, en le faisant suivre du nom d'une variable existante (déclarée précédemment). Par exemple :

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

Spécifie que la variable *StrLen* doit commencer à la même adresse que *Str*. Comme le premier octet d'une chaîne courte contient la longueur de la chaîne, la valeur de *StrLen* est la longueur de *Str*.

Il n'est pas possible d'initialiser une variable dans une déclaration **absolute**, ni de combiner **absolute** avec d'autres directives.

Variables dynamiques

Il est possible de créer des variables dynamiques en appelant la procédure *GetMem* ou *New*. De telles variables sont allouées sur le tas et ne sont pas gérées automatiquement. Quand vous en créez une, c'est à vous de libérer la mémoire de la variable : utilisez *FreeMem* pour supprimer des variables créées par *GetMem* et *Dispose* pour supprimer des variables créées par *New*. D'autres routines standard agissent sur les variables dynamiques : *ReallocMem*, *Initialize*, *StrAlloc* et *StrDispose*.

Les chaînes longues, les chaînes étendues, les tableaux dynamiques, les variants et les interfaces sont également des variables dynamiques allouées sur la pile, mais leur mémoire est gérée automatiquement.

Variables locales de thread

Les variables locales de thread (appelées aussi *variables de thread*) sont utilisées dans les applications multithreads. Une variable locale de thread est identique à une variable globale à cette différence que chaque thread d'exécution dispose de sa propre copie privée de la variable à laquelle les autres threads ne peuvent accéder. Les variables locales de thread sont déclarées avec le mot réservé **threadvar** au lieu de **var**. Par exemple :

```
threadvar X: Integer;
```

Les déclarations de variables de thread

- ne peuvent se faire dans une procédure ou une fonction.
- ne peuvent contenir d'initialisation.
- ne peuvent spécifier la directive **absolute**.

Ne créez pas de variables de thread de type procédure ou pointeur et n'utilisez pas de variables de thread dans les bibliothèques à chargement dynamique (autres que les paquets).

Les variables dynamiques qui sont généralement gérées par le compilateur — chaînes longues, chaînes étendues, tableaux dynamiques, variants et interfaces — peuvent êtres déclarées avec **threadvar**, mais le compilateur ne libère pas automatiquement la mémoire allouée sur le tas créée par chaque thread d'exécution. Si vous utilisez ces types de données dans des variables de thread, c'est à vous de vous occuper de leur mémoire. Par exemple,

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
 ⋮
S := '';  // libère la mémoire utilisée par S
```

(Vous pouvez libérer un variant en le définissant par *Unassigned* et une interface ou un tableau dynamique en le définissant par **nil**.)

Constantes déclarées

Différentes constructions du langage sont désignées par le terme “constantes”. Il y a d'une part les constantes numériques (des *nombres*) comme `17` et des constantes chaînes (appelées également *chaînes de caractères* ou *littéraux chaîne*) comme `'Bonjour chez vous!'` ; pour davantage d'informations sur les constantes numériques et chaîne, voir chapitre 4, “Eléments syntaxiques”. Chaque type énuméré définit des constantes qui représentent les valeurs de ce type. Il y a aussi des constantes prédéfinies comme *True*, *False* ou **nil**. Enfin, il y a des constantes qui, comme les variables, sont créées individuellement par des déclarations.

Les constantes déclarées sont soit de *vraies constantes,* soit des *constantes typées*. Ces deux sortes de constantes sont apparemment similaires, mais elles obéissent à des règles différentes et s'emploient pour des usages différents.

Vraies constantes

Une vraie constante est un identificateur déclaré dont la valeur ne peut changer. Par exemple :

```
const ValeurMax = 237;
```

déclare une constante appelée *ValeurMax* qui renvoie l'entier 237. La syntaxe de déclaration d'une vraie constante est :

const *identificateur* = *expressionConstante*

où *identificateur* est un identificateur valide et *expressionConstante* est une expression qui peut être évaluée par le compilateur sans exécuter le programme. Pour davantage d'informations, voir "Expressions constantes" à la page 5-45.

Si *expressionConstante* renvoie une valeur scalaire, vous pouvez spécifier le type de la constante déclarée en utilisant un transtypage de valeur. Par exemple :

```
const UnNombre = Int64(17);
```

déclare une constante appelée *UnNombre*, de type *Int64*, qui renvoie l'entier 17. Sinon, le type de la constante déclarée est du type de *expressionConstante*.

- Si *expressionConstante* est une chaîne de caractères, la constante déclarée est compatible avec tout type de chaîne. Si la chaîne de caractères est de longueur 1, elle est également compatible avec tout type de caractère.
- Si *expressionConstante* est un réel, son type est *Extended*. Si c'est un entier, son type est donnée par le tableau suivant.

Tableau 5.8 Types des constantes entières

Etendue de la constante (hexadécimal)	Etendue de la constante (décimal)	Type
–$8000000000000000..–$80000001	-2^{63}..–2147483649	*Int64*
–$80000000..–$8001	–2147483648..–32769	*Integer*
–$8000..–$81	–32768..–129	*Smallint*
–$80..–1	–128..–1	*Shortint*
0..$7F	0..127	0..127
$80..$FF	128..255	*Byte*
$0100..$7FFF	256..32767	0..32767
$8000..$FFFF	32768..65535	*Word*
$10000..$7FFFFFFF	65536..2147483647	0..2147483647
$80000000..$FFFFFFFF	2147483648..4294967295	*Cardinal*
$100000000..$7FFFFFFFFFFFFFFF	4294967296..$2^{63}-1$	*Int64*

Voici quelques exemples de déclarations de constantes :

```
const
  Min = 0;
  Max = 100;
  Centre = (Max - Min) div 2;
```

```
Beta = Chr(225);
NbCars = Ord('Z') - Ord('A') + 1;
Message = 'Mémoire insuffisante';
ErrStr = ' Erreur : ' + Message + '. ';
ErrPos = 80 - Length(ErrStr) div 2;
Ln10 = 2.302585092994045684;
Ln10R = 1 / Ln10;
Numerique = ['0'..'9'];
Alpha = ['A'..'Z', 'a'..'z'];
AlphaNum = Alpha + Numerique;
```

Expressions constantes

Une *expression constante* est une expression que le compilateur peut évaluer sans exécuter le programme dans lequel se trouve la déclaration. Les expressions constantes sont : les nombres, les chaînes de caractères, les vraies constantes, les valeurs des types énumérés, les constantes spéciales *True*, *False* et **nil** ; et toutes les expressions construites exclusivement en utilisant ces éléments avec des opérateurs, des conversions de type et des constructeurs d'ensemble. Les expressions constantes ne peuvent contenir de variables, de pointeurs ou d'appels de fonction, sauf les fonctions prédéfinies suivantes :

Abs	*High*	*Low*	*Pred*	*Succ*
Chr	*Length*	*Odd*	*Round*	*Swap*
Hi	*Lo*	*Ord*	*SizeOf*	*Trunc*

Cette définition d'une *expression constante* intervient à plusieurs endroits dans la spécification de la syntaxe Pascal Objet. Les expressions constantes sont nécessaires à l'initialisation des variables globales, la définition de types intervalle, l'affectation de rang aux valeurs des types énumérés, la spécification de la valeur par défaut de paramètres, l'écriture d'instructions **case** et la déclaration de vraies constantes et de constantes typées.

Voici des exemples d'expressions constantes :

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Développeur'
Chr(32)
Ord('Z') - Ord('A') + 1
```

Chaînes de ressource

Les chaînes de ressource sont stockées comme des ressources et liées à l'exécutable ou la bibliothèque afin de pouvoir les modifier sans avoir besoin de recompiler le programme. Pour davantage d'informations, voir les rubriques de l'aide en ligne traitant de la localisation des applications.

Les chaînes de ressource sont déclarées comme les vraies constantes, à cette différence que le mot **const** est remplacé par **resourcestring**. L'expression à

droite du symbole = doit être une expression constante qui renvoie une valeur chaîne. Par exemple :

```
resourcestring
  CreateError = 'Impossible de créer le fichier %s';  {  pour des informations sur les
                                                          spécificateurs de format, }
  OpenError = 'Impossible d''ouvrir le fichier %s';   {  voir 'chaînes de format' dans
                                                          l'aide en ligne }
  LineTooLong = 'Ligne trop longue';
  ProductName = 'Borland Rocks\000\000';
  SomeResourceString = UneVraieConstante;
```

Le compilateur résoud automatiquement les conflits de nom entre les chaînes de ressource de différentes bibliothèques.

Constantes typées

Les constantes typées, à la différences des vraies constantes, peuvent contenir des valeurs de type tableau, enregistrement, procédure ou pointeur. Les constantes typées ne peuvent intervenir dans des expressions constantes.

Dans l'état par défaut du compilateur **{$J+}**, il est même possible d'affecter de nouvelles valeurs à des constantes typées : elles se comportent alors essentiellement comme des variables initialisées. Mais si la directive de compilation **{$J–}** est active, il n'est pas possible à l'exécution de modifier la valeur des constantes typées ; en effet ce sont alors des variables en lecture seule.

Déclarez une constante typée de la manière suivante :

const *identificateur*: *type* = *valeur*

où *identificateur* est un identificateur valide, *type* est un type quelconque (sauf un type fichier ou variant) et *valeur* est une expression de type *type*. Par exemple,

```
const Max: Integer = 100;
```

Le plus souvent, *valeur* doit être une expression constante ; mais si *type* est un type tableau, enregistrement, procédure ou pointeur, des règles spéciales s'appliquent.

Constantes tableau

Pour déclarer une constante tableau, placez entre parenthèses à la fin de la déclaration les valeurs des éléments du tableau qui sont séparées par des virgules. Ces valeurs doivent être représentées par des expressions constantes.

Par exemple :

```
const Chiffres: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

déclare une constante typée appelée *Chiffres* qui contient un tableau de caractères.

Les tableaux de caractères d'indice de base zéro représentent souvent des chaînes à zéro terminal, les constantes chaînes peuvent donc être utilisées pour initialiser

des tableaux de caractères. Ainsi, la déclaration précédente peut se formuler plus simplement par :

```
const Chiffres: array[0..9] of Char = '0123456789';
```

Pour définir une constante tableau à plusieurs dimensions, placez séparément entre parenthèses les valeurs de chaque dimension, séparez les dimensions par des virgules. Par exemple :

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Labyrinthe : TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

crée un tableau appelé *Labyrinthe* où :

Labyrinthe[0,0,0] = 0
Labyrinthe[0,0,1] = 1
Labyrinthe[0,1,0] = 2
Labyrinthe[0,1,1] = 3
Labyrinthe[1,0,0] = 4
Labyrinthe[1,0,1] = 5
Labyrinthe[1,1,0] = 6
Labyrinthe[1,1,1] = 7

A aucun niveau, les constantes tableau ne peuvent contenir de valeurs de type fichier.

Constantes enregistrement

Pour déclarer une constante enregistrement, spécifiez entre parenthèses la valeur des champs en séparant les affectations de champ (de la forme *identificateurChamp*: *valeur*) par des points-virgules. Les valeurs doivent être représentées par des expressions constantes. Les champs doivent être énumérés dans l'ordre de leur déclaration dans la déclaration du type enregistrement. S'il existe, la valeur du champ sélecteur doit être spécifiée ; si l'enregistrement a une partie variable, il faut assigner des valeurs uniquement à la variante sélectionnée par le champ sélecteur.

Exemples :

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVecteur = array[0..1] of TPoint;
  TMois = (Jan, Fev, Mar, Avr, Mai, Jun, Jul, Aou, Sep, Oct, Nov, Dec);
  TDate = record
    J: 1..31;
    M: TMois;
    A: 1900..1999;
  end;
const
  Origine: TPoint = (X: 0.0; Y: 0.0);
  Ligne: TVecteur = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  UnJour: TDate = (J: 2; M: Dec; A: 1960);
```

A aucun niveau, les constantes enregistrement ne peuvent contenir de valeur de type fichier.

Constantes procédure

Pour déclarer une constante procédure, spécifiez le nom d'une fonction ou d'une procédure compatible avec le type déclaré de la constante. Par exemple :

```
function Calc(X, Y: Integer): Integer;
begin
 :
end;

type TFonction = function(X, Y: Integer): Integer;
const MaFonction: TFunction = Calc;
```

Etant donné ces déclarations, vous pouvez utiliser la constante procédure *MaFonction* dans un appel de fonction :

```
I := MaFonction(5, 7)
```

Vous pouvez également affecter la valeur **nil** à une constante procédure.

Constantes pointeur

Quand vous déclarez une constante pointeur, vous devez l'initialiser avec une valeur qui peut être résolue à la compilation (au moins sous la forme d'une adresse relative). Il y a trois manières de procéder pour ce faire : avec l'opérateur **@**, avec **nil** et (si la constante est de type *PChar*) avec un littéral chaîne. Si, par exemple, *I* est une variable globale de type *Integer*, vous pouvez déclarer une constante de la forme

```
const PI: ^Integer = @I;
```

Le compilateur peut résoudre cette expression car les variables globales font partie du segment de code. Tout comme les fonctions et les constantes globales :

```
const PF: Pointer = @MaFonction;
```

Comme les littéraux chaîne sont alloués comme des constantes globales, vous pouvez initialiser une constante *PChar* avec un littéral chaîne :

```
const WarningStr: PChar = 'Attention!';
```

Les adresses des variables locales (allouées dans la pile) et dynamiques (allouées dans le tas) ne peuvent être affectées à des constantes pointeur.

Chapitre

6

Procédures et fonctions

Les procédures et fonctions, désignées collectivement par le terme *routine*, sont des blocs d'instructions autonomes qui peuvent être appelés depuis divers endroits d'un programme. Une *fonction* est une routine qui renvoie une valeur quand elle est exécutée. Une *procédure* est une routine qui ne renvoie pas de valeur.

Les appels de fonction peuvent être utilisés comme expression dans les affectations et les opérations car ils renvoient une valeur. Par exemple :

```
I := UneFonction(X);
```

appelle *UneFonction* et affecte son résultat à *I*. Les appels de fonction ne peuvent pas apparaître du côté gauche d'une instruction d'affectation.

Les appels de procédures —et, quand la syntaxe étendue est activée, (**{$X+}**), les appels de fonctions peuvent s'utiliser comme des instructions à part entière. Par exemple :

```
FaireQuelquechose;
```

appelle la routine *FaireQuelquechose* ; si *FaireQuelquechose* est une fonction, la valeur renvoyée est perdue.

Les procédures et fonctions peuvent s'appeler elles-mêmes de manière récursive.

Déclaration de procédures et de fonctions

Quand vous déclarez une procédure ou une fonction, vous spécifiez son nom, le nombre et le type des paramètres et, dans le cas d'une fonction, le type de la valeur renvoyée. Cette partie de la déclaration est parfois appelée le *prototype* ou *l'en-tête* de la routine. Puis, vous écrivez un bloc de code qui s'exécute à chaque fois que la procédure ou la fonction est appelée. Cette partie est parfois appelée le *corps* ou le *bloc* de la routine.

La procédure standard *Exit* peut apparaître dans le corps de toute routine. *Exit* interrompt l'exécution de la routine à l'endroit de son appel et restitue le contrôle du programme au point d'appel de la routine.

Déclaration de procédures

La déclaration d'une procédure a la forme :

```
procedure nomProcédure (listeParamètres); directives;
déclarationsLocales;
begin
  instructions
end;
```

où *nomProcédure est un identificateur* valide, *instructions* une série d'instructions qui s'exécute quand la procédure est appelée. Les éléments (*listeParamètres*), *directives*; et *déclarationsLocales*; sont facultatifs.

- Pour des informations sur la *listeParamètres*, voir "Paramètres" à la page 6-10.
- Pour des informations sur les *directives*, voir "Conventions d'appel" à la page 6-5, "Déclaration forward et interface" à la page 6-6, "Déclarations externes" à la page 6-7, "Redéfinition de procédures et de fonctions" à la page 6-8 et "Ecriture de bibliothèques à chargement dynamique" à la page 9-4. Si vous spécifiez plusieurs directives, séparez-les par des points-virgules.
- Pour des informations sur les *déclarationsLocales*, qui déclarent des identificateurs locaux, voir "Déclarations locales" à la page 6-10.

Voici un exemple de déclaration de procédure :

```
procedure NbChaine(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

Etant donnée cette déclaration, vous pouvez appeler la procédure *NbChaine* de la manière suivante :

```
NBChaine(17, MaChaine);
```

Cette procédure affecte la valeur "17" à *MaChaine* (qui doit être une variable **string**).

Dans le bloc d'instructions d'une procédure, vous pouvez utiliser des variables et d'autres identificateurs déclarés dans la partie *déclarationsLocales* de la procédure. Vous pouvez également utiliser les noms de paramètre de la liste de paramètres

(comme *N* et *S* dans l'exemple précédent). La liste de paramètres définit un ensemble de variables locales, vous ne devez donc pas redéclarer le nom des paramètres dans la section *déclarationsLocales*. Vous pouvez, enfin, utiliser tous les identificateurs qui sont dans la portée de la déclaration de la procédure.

Déclaration de fonctions

Une déclaration de fonction est similaire à la déclaration d'une procédure mais elle spécifie le type de la valeur renvoyé. Une déclaration de fonction a la forme suivante :

```
function nomFonction(listeParamètres): typeRenvoyé; directives;
déclarationsLocales;
begin
  instructions
end;
```

où *nomFonction est un identificateur* valide, *typeRenvoyé* est un nom de type, *instructions* la série des instructions exécutées quand la fonction est appelée. Les éléments (*listeParamètres*), *directives*; et *déclarationsLocales*; sont facultatifs.

- Pour des informations sur la *listeParamètres*, voir “Paramètres” à la page 6-10.
- Pour des informations sur les *directives*, voir “Conventions d'appel” à la page 6-5, “Déclaration forward et interface” à la page 6-6, “Déclarations externes” à la page 6-7, “Redéfinition de procédures et de fonctions” à la page 6-8 et “Ecriture de bibliothèques à chargement dynamique” à la page 9-4. Si vous spécifiez plusieurs directives, séparez-les par des points-virgules.
- Pour des informations sur les *déclarationsLocales*, qui déclarent des identificateurs locaux, voir “Déclarations locales” à la page 6-10.

Le bloc instruction d'une fonction respecte les mêmes règles que celles qui s'appliquent aux procédures. A l'intérieur du bloc instruction, vous pouvez utiliser les variables et les autres identificateurs déclarés dans la partie *déclarationsLocales* de la fonction, les noms de paramètre de la liste de paramètres et les identificateurs dont la portée est dans la déclaration de la fonction. De plus, le nom de la fonction se comporte comme une variable spéciale contenant la valeur renvoyée par la fonction, de même que la variable prédéfinie *Result*.

Par exemple :

```
function WF: Integer;
begin
  WF := 17;
end;
```

définit une fonction constante appelée *WF* qui n'attend pas de paramètre et renvoie toujours la valeur entière 17. Cette déclaration est équivalent à :

```
function WF: Integer;
begin
  Result := 17;
end;
```

Voici un exemple de déclaration de fonction plus compliquée :

```
function Max(A: array of Real; N: Integer): Real;
var
  X: Real;
  I: Integer;
begin
  X := A[0];
  for I := 1 to N - 1 do
    if X < A[I] then X := A[I];
  Max := X;
end;
```

Dans le bloc instruction, vous pouvez affecter plusieurs fois une valeur à *Result* ou au nom de la fonction ; il faut simplement que le type de la valeur affectée corresponde au type renvoyé déclaré. Quand l'exécution de la fonction s'achève, la dernière valeur affectée à *Result* ou au nom de la fonction définit la valeur renvoyée par la fonction. Par exemple :

```
function Puissance(X: Real; Y: Integer): Real;
var
  I: Integer;
begin
  Result := 1.0;
  I := Y;
  while I > 0 do
  begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
  end;
end;
```

Result et le nom de la fonction représentent toujours la même valeur. Ainsi la fonction :

```
function MaFonction: Integer;
begin
  MaFonction := 5;
  Result := Result * 2;
  MaFonction := Result + 1;
end;
```

renvoie la valeur 11. Cependant *Result* n'est pas complètement interchangeable avec le nom de la fonction. Quand le nom de la fonction apparaît à gauche d'une instruction d'affectation, le compilateur suppose qu'il est utilisé (comme *Result*) pour indiquer la valeur renvoyée. Par contre, quand le nom de la fonction apparaît n'importe où ailleurs dans le bloc instruction, le compilateur l'interprète comme un appel récursif de la fonction. Par contre, *Result* s'utilise comme une variable dans les opérations, les transtypages, les constructeurs d'ensemble, les indices et les appels à d'autres procédures.

Si la syntaxe étendue est activée (**{$X+}**), *Result* est déclarée implicitement dans chaque fonction. Vous ne devez donc pas la redéclarer. Si l'exécution s'achève sans qu'une valeur soit affectée à *Result* ou au nom de la fonction, la valeur renvoyée par la fonction est indéfinie.

Conventions d'appel

Dans la déclaration d'une procédure ou d'une fonction, vous pouvez spécifier une *convention d'appel* en utilisant l'une des directives **register**, **pascal**, **cdecl**, **stdcall** et **safecall**. Par exemple :

```
function MaFonction(X, Y: Real): Real; cdecl;
  ⋮
```

Les conventions d'appel déterminent l'ordre dans lequel les paramètres sont transmis à la routine. Elles affectent également la suppression des paramètres de la pile, l'utilisation de registres pour transmettre les paramètres, ainsi que la gestion des erreurs et des exceptions. **register** est la convention d'appel par défaut.

- Les conventions **register** et **pascal** transmettent les paramètres de gauche à droite ; c'est-à-dire que le paramètre de gauche est évalué et transmis en premier, le paramètre de droite est évalué et transmis en dernier. Les conventions **cdecl**, **stdcall** et **safecall** transmettent les paramètres de droite à gauche.
- Pour toutes les conventions à l'exception de **cdecl**, les procédures et fonctions suppriment les paramètres de la pile lors de la sortie. Avec la convention **cdecl**, c'est à l'appelant de supprimer les paramètres de la pile au retour de l'appel.
- La convention **register** utilise jusqu'à trois registres de la CPU pour transmettre des paramètres alors que toutes les autres conventions transmettent tous les paramètres dans la pile.
- La convention **safecall** implémente les "coupe-feu" d'exceptions. Sous Windows, cela implémente la notification d'erreurs COM interprocessus.

Le tableau suivant résume les caractéristiques des conventions d'appel.

Tableau 6.1 Conventions d'appel

Directive	Ordre des paramètres	Nettoyage	Transfert de paramètre dans les registres?
register	De gauche à droite	Routine	Oui
pascal	De gauche à droite	Routine	Non
cdecl	De droite à gauche	Appelant	Non
stdcall	De droite à gauche	Routine	Non
safecall	De droite à gauche	Routine	Non

La convention d'appel par défaut **register** est la plus efficace car elle évite la création d'un cadre de pile. (Les méthodes d'accès aux propriétés publiées doivent utiliser **register**.) La convention **cdecl** est utile pour les appels de fonctions à partir de bibliothèques partagées écrites en C ou en C++, alors que **stdcall** et **safecall** sont conservées habituellement pour les appels à du code externe. Sous Windows, les API du système d'exploitation sont **stdcall** et **safecall**. Les autres systèmes d'exploitation utilisent généralement **cdecl**. (Notez que **stdcall** est plus efficace que **cdecl**.)

La convention **safecall** doit être utilisée pour déclarer les méthodes des interfaces doubles (voir chapitre 10, “Interfaces d’objets”). La convention **pascal** est conservée dans un souci de compatibilité ascendante. Pour davantage d’informations sur les conventions d’appel, voir chapitre 12, “Contrôle des programmes”.

Les directives **near**, **far** et **export** désignent des conventions d’appels utilisées dans la programmation Windows 16 bits. Elles sont sans effet dans les applications 32 bits et sont conservées uniquement dans un souci de compatibilité ascendante.

Déclaration forward et interface

Dans une déclaration de procédure ou de fonction, la directive **forward** se substitue au bloc, y compris à la déclaration des variables locales et des instructions. Par exemple :

```
function Calcul(X, Y: Integer): Real; forward;
```

déclare une fonction appelée *Calcul*. Quelque part après la déclaration **forward**, la routine doit être redéclarée dans une *déclaration de définition* qui contient un bloc. La déclaration de définition de la fonction *Calcul* peut être :

```
function Calcul;
 ⋮   { déclarations }
begin
 ⋮   { bloc instruction}
end;
```

Généralement, une déclaration de définition ne répète pas la liste des paramètres ou le type renvoyé par la routine. Mais si vous les répétez, ils doivent correspondre exactement à ceux faits dans la déclaration **forward** (à cette différence que les paramètres par défaut peuvent être omis). Si la déclaration **forward** spécifie une procédure ou une fonction redéfinie (voir “Redéfinition de procédures et de fonctions” à la page 6-8), la déclaration de définition doit alors répéter la liste des paramètres.

Entre la déclaration **forward** et sa déclaration de définition, vous ne pouvez rien placer si ce n’est d’autres déclarations. La déclaration de définition peut être une déclaration **external** ou **assembler**, mais pas une autre déclaration **forward**.

Le rôle d’une déclaration **forward** est d’étendre en avant dans le code la portée d’un identificateur de routine. Cela permet à d’autres procédures et fonctions d’appeler la routine déclarée **forward** avant qu’elle ne soit effectivement définie. En dehors du fait que cela permet d’organiser le code de manière plus souple, les déclarations **forward** sont parfois indispensables dans le cas de récursions mutuelles.

La directive **forward** n’est pas autorisée dans la section **interface** d’une unité. En effet, les en-têtes de procédures et de fonctions de la section **interface** se comportent comme des déclarations **forward** dont la déclaration de définition doit se trouver dans la section **implementation**. Une routine déclarée dans la

section **interface** est utilisable partout ailleurs dans l'unité et dans toutes les unités et programmes utilisant l'unité où elle est déclarée.

Déclarations externes

La directive **external** qui remplace le bloc dans une déclaration de procédure ou de fonction permet d'appeler des routines compilées séparément de votre programme. Les routines externes peuvent être issues de fichiers objet ou de bibliothèques à chargement dynamique.

Quand vous importez une fonction C++ qui prend un nombre variable de paramètres, utilisez la directive **varargs**. Par exemple,

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

La directive **varargs** fonctionne uniquement avec des routines externes et uniquement avec la convention d'appel **cdecl**.

Liaison de fichiers objet

Pour appeler des routines d'un fichier objet compilé séparément, commencez par lier le fichier objet à votre application en utilisant la directive de compilation **$L** (ou **$LINK**). Par exemple :

```
Pour Windows : {$L BLOCK.OBJ}

Pour Linux :   {$L block.o}
```

lie BLOCK.OBJ (Windows) ou block.o (Linux) au programme ou à l'unité dans laquelle cette directive est placée. Ensuite, déclarez les fonctions et procédures que vous voulez appeler :

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Vous pouvez ensuite appeler les routines *MoveWord* et *FillWord* depuis BLOCK.OBJ (Windows) ou block.o (Linux).

De telles déclarations sont fréquemment utilisées pour accéder à des routines externes écrites en assembleur. Vous pouvez aussi placer directement des routines en assembleur dans votre code source Pascal Objet ; pour davantage d'informations, voir chapitre 13, "Code assembleur en ligne".

Importation des fonctions de bibliothèques

Pour importer des routines d'une bibliothèque à chargement dynamique, attachez une directive de la forme :

```
external constanteChaîne;
```

à la fin de l'en-tête de la fonction ou de la procédure, où *constanteChaîne* est le nom du fichier bibliothèque placé entre apostrophes. Par exemple, sous Windows :

```
function UneFonction(S: string): string; external 'strlib.dll';
```

importe de strlib.dll une fonction appelée *UneFonction*.

Sous Linux,

```
function UneFonction(S: string): string; external 'strlib.so';
```

importe de strlib.so une fonction appelée *UneFonction*.

Vous pouvez importer une routine sous un nom différent de celui qu'elle a dans la bibliothèque. Si vous le faites, spécifiez le nom original dans la directive **external** :

external *constanteChaîne*$_1$ name *constanteChaîne*$_2$;

où la première *constanteChaîne* spécifie le nom du fichier bibliothèque et la seconde *constanteChaîne* est le nom original de la routine.

Sous Windows : Par exemple la déclaration suivante importe une fonction depuis user32.dll (partie de l'API Windows).

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer;
  stdcall; external 'user32.dll' name 'MessageBoxA';
```

Le nom d'origine de la fonction est *MessageBoxA*, mais elle est importée sous le nom *MessageBox*.

A la place du nom, vous pouvez utiliser un numéro pour identifier la routine à importer :

external *constanteChaîne* index *constanteEntière*;

où *constanteEntière* et l'indice de la routine dans la table d'exportation.

Sous Linux : Par exemple la déclaration suivante importe une fonction système standard de libc.so.6.

```
function OpenFile(const PathName: PChar; Flags: Integer): Integer; cdecl;
  external 'libc.so.6' name 'open';
```

Le nom d'origine de la fonction est *open* mais elle est importée sous le nom *OpenFile*.

Dans la déclaration d'importation, assurez-vous de respecter exactement l'orthographe et la casse du nom de la routine. Par contre, une fois la routine importée, il n'y a plus de différences majuscules/minuscules.

Pour davantage d'informations sur les bibliothèques, voir chapitre 9, "Bibliothèques et paquets".

Redéfinition de procédures et de fonctions

Il est possible de redéclarer plusieurs fois une routine dans la même portée sous le même nom. C'est ce que l'on appelle la *redéfinition* (ou *surcharge*). Les routines redéfinies doivent être redéclarées avec la directive **overload** et doivent utiliser une liste de paramètres différente. Soit, par exemple, les déclarations :

```
function Diviser(X, Y: Real): Real; overload;
begin
  Result := X/Y;
end;
```

```
function Diviser(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

Ces déclarations créent deux fonctions appelées toutes les deux *Diviser* qui attendent des paramètres de types différents. Quand vous appelez *Diviser*, le compilateur détermine la fonction à utiliser en examinant les paramètres effectivement transmis dans l'appel. Ainsi, `Diviser(6.0, 3.0)` appelle la première fonction *Diviser* car ses arguments sont des valeurs réelles.

Vous pouvez transmettre à une routine redéfinie des paramètres qui ne sont pas du même type que ceux d'une des déclarations de la routine, mais qui sont compatibles au niveau de l'affectation avec des paramètres d'une ou de plusieurs déclarations. Cela arrive le plus souvent quand une routine est redéfinie avec différents types entiers ou différents types réels — par exemple,

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

Dans ce cas, quand c'est possible de le faire sans ambiguité, le compilateur appelle la routine dont les paramètres sont du type de la plus petite étendue qui convienne aux paramètres réels de l'appel. (N'oubliez pas que les expressions de constantes réelles sont toujours de type *Extended*.)

Les routines redéfinies doivent pouvoir se distinguer par le nombre ou le type de leurs paramètres. Ainsi, la paire de déclarations suivante déclenche une erreur de compilation :

```
function Maj(S: string): string; overload;
  ⋮
procedure Maj(var Str: string); overload;
  ⋮
```

Alors que les déclarations :

```
function Fonc(X: Real; Y: Integer): Real; overload;
  ⋮
function Fonc(X: Integer; Y: Real): Real; overload;
  ⋮
```

sont légales.

Quand une routine redéfinie est déclarée dans une déclaration **forward** ou d'interface, la déclaration de définition doit obligatoirement répéter la liste des paramètres de la routine.

Si vous utilisez des paramètres par défaut dans des routines redéfinies, méfiez-vous des signatures de paramètres ambiguës. Pour davantage d'informations, voir "Paramètres par défaut et routines redéfinies" à la page 6-19.

Vous pouvez limiter les effets potentiels de la redéfinition en qualifiant le nom d'une routine lors de son appel. Par exemple, `Unit1.MaProcedure(X, Y)` n'appelle que les routines déclarées dans *Unit1* ; si aucune routine de *Unit1* ne correspond au nom et à la liste des paramètres, il y a une erreur de compilation.

Pour des informations sur la distribution de méthodes redéfinies dans une hiérarchie de classes, voir “Redéfinition de méthodes” à la page 7-13. Pour des informations sur l’exportation de routines redéfinie depuis une bibliothèque partagée, voir “Clause exports” à la page 9-5.

Déclarations locales

Le corps d’une fonction ou d’une procédure commence souvent par la déclaration de variables locales utilisées dans le bloc instruction de la routine. Ces déclarations peuvent également contenir des constantes, des types ou d’autres routines. La portée d’un identificateur local est limitée à celle de la routine dans laquelle il est déclaré.

Routines imbriquées

Les fonctions et procédures contiennent parfois d’autres fonctions ou procédures dans la section des déclarations locales de leur bloc. Par exemple le code suivant déclare une procédure *FaireQuelquechose* qui contient une procédure imbriquée :

```
procedure FaireQuelquechose(S: string);
var
  X, Y: Integer;

  procedure ProcImbriquee(S: string);
  begin
    ⋮
  end;

begin
   ⋮
  ProcImbriquee(S);
   ⋮
end;
```

La portée d’une routine imbriquée est limitée à la fonction ou la procédure dans laquelle elle est déclarée. Dans l’exemple précédent, *ProcImbriquee* ne peut être appelée que dans *FaireQuelquechose*.

Pour des exemples réels de procédures imbriquées, examinez la procédure *DateTimeToString*, la fonction *ScanDate* et d’autres routines de l’unité *SysUtils*.

Paramètres

La plupart des en-têtes de procédure et de fonction comportent une *liste de paramètres*. Par exemple, dans l’en-tête :

```
function Puissance(X: Real; Y: Integer): Real;
```

la liste de paramètres est `(X: Real; Y: Integer)`.

Une liste de paramètres est une suite placée entre parenthèses de déclarations de paramètres séparées par des points-virgules. Chaque déclaration est une liste séparée par des virgules de noms de paramètres suivi le plus souvent du

symbole deux points et d'un identificateur de type et, dans certains cas, du symbole = et d'une valeur par défaut. Les noms de paramètre doivent être des identificateurs valides. Chaque déclaration peut être précédée par un des mots réservés **var**, **const et out**. Exemples :

```
(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Texte, Libelle: PChar; Indicateurs: Integer)
(const P; I: Integer)
```

La liste des paramètres spécifie le nombre, l'ordre et le type des paramètres qui doivent être transmis à la routine lors de son appel. Si une routine n'utilise aucun paramètre, omettez dans sa déclaration la liste d'identificateurs et les parenthèses :

```
procedure ActualiserEnregs;
begin
 ⋮
end;
```

A l'intérieur du corps de la procédure ou de la fonction, les noms de paramètres (*X* et *Y* dans le premier exemple ci-dessus) peuvent être utilisés comme des variables locales. Ne redéclarez pas le nom des paramètres dans la section des déclarations locales du corps de la routine.

Sémantique des paramètres

Les paramètres peuvent être catégorisés de différentes manières :

- Chaque paramètre est classé comme paramètre *valeur*, *variable*, *constante* ou *de sortie*. Les paramètres sont par défaut des paramètres valeur ; les mots réservés **var**, **const** et **out** indiquent, respectivement des paramètres variables, constantes et de sortie.
- Les paramètres valeur sont toujours *typés*, alors que les paramètres constantes, variables et de sortie peuvent être typés ou *sans type*.
- Des règles spéciales s'appliquent aux paramètres tableau. Voir “Paramètres chaîne” à la page 6-14.

Les fichiers et les instances de types structurés contenant des fichiers ne peuvent être transmis que comme paramètres variables (**var**).

Paramètres valeur et paramètres variables

La plupart des paramètres sont des paramètres valeur (c'est le cas par défaut) ou des paramètres variables (**var**). Les paramètres valeur sont transmis *par valeur* alors que les paramètres variables sont transmis *par adresse*. Pour saisir la différence, examinez les fonctions suivantes :

```
function DoubleParValeur(X: Integer): Integer;     // X est un paramètre valeur
begin
  X := X * 2;
  Result := X;
end;
```

```
function DoubleParAdresse(var X: Integer): Integer;   // X est un paramètre variable
begin
  X := X * 2;
  Result := X;
end;
```

Ces deux fonctions renvoient le même résultat, mais seule la seconde (*DoubleParAdresse)* peut modifier la valeur de la variable qui lui est transmise. Si les deux fonctions sont appelées de la manière suivante :

```
var
  I, J, V, W: Integer;
begin
  I := 4;
  V := 4;
  J := DoubleParValeur(I);      // J = 8, I = 4
  W := DoubleParAdresse(V);     // W = 8, V = 8
end;
```

Après l'exécution de ce code, la variable *I*, transmise à *DoubleParValeur*, garde la même valeur que celle qui lui a été affectée initialement. Mais la valeur de la variable *V*, qui a été transmise à *DoubleParAdresse* est, elle, modifiée.

Un paramètre valeur se comporte comme une variable locale qui est initialisée avec la valeur transmise à l'appel de la routine. Si vous transmettez une variable comme paramètre valeur, la routine en crée une copie et les modifications apportées à la copie sont sans effet sur la variable d'origine et sont perdues quand l'exécution du programme revient à l'appel de la routine.

Un paramètre variable se comporte, lui, comme un pointeur et non comme une copie. Les modifications apportées aux paramètres dans le corps de la routine sont conservées lorsque l'exécution du programme revient à l'appel de la routine et que le nom du paramètre est hors de portée.

Il n'y a pas de copie effectuée même quand la même variable est transmise à plusieurs paramètres **var**. Ce comportement est mis en évidence par l'exemple suivant :

```
procedure AjouterUn(var X, Y: Integer);
begin
  X := X + 1;
  Y := Y + 1;
end;

var I: Integer;
begin
  I := 1;
  AjouterUn(I, I);
end;
```

Après l'exécution de ce code, *I* a la valeur 3.

Si la déclaration d'une routine spécifie un paramètre **var**, vous devez, lors de l'appel de la routine, lui transmettre une expression affectable (c'est-à-dire une variable, une constante typée (dans l'état **{$J+}**), un pointeur déréférencé, un champ, une variable indicée). Ainsi, dans l'exemple ci-dessus, `DoubleParAdresse(7)` provoque une erreur de compilation alors que `DoubleParValeur(7)` est légal.

Les indices et les pointeurs de référence transmis dans des paramètres **var**, par exemple `DoubleParAdresse(MyArray[I])`, ne sont évalués qu'une seule fois, avant l'exécution de la routine.

Paramètres constantes

Un paramètre constante (**const**) est semblable à une constante locale ou à une variable en lecture seule. Les paramètres constantes sont semblables aux paramètres par valeur à cette différence qu'il n'est pas possible d'affecter une valeur à un paramètre constante dans le corps de la routine, ni de le transmettre comme paramètre **var** à une autre routine. Par contre, si vous transmettez une référence d'objet comme paramètre constante, il est quand même possible de modifier la valeur des propriétés de l'objet.

L'utilisation de **const** permet au compilateur d'optimiser le code pour les paramètres de type structuré ou chaîne. Cela évite également de transmettre involontairement par adresse le paramètre à une autre routine.

Voici, par exemple, l'en-tête de la fonction *CompareStr* de l'unité *SysUtils* :

```
function CompareStr(const S1, S2: string): Integer;
```

Comme *S1* et *S2* ne sont pas modifiés dans le corps de *CompareStr*, ils peuvent être déclarés comme paramètres constantes.

Paramètres de sortie

Un paramètre de sortie (**out**) est transmis par adresse comme un paramètre variable. Mais avec un paramètre **out**, la valeur initiale de la variable référencée n'est pas prise en compte par la routine à laquelle elle est transmise. Le paramètre **out** n'est utilisé qu'en sortie ; il indique simplement à la routine où placer la valeur en sortie sans spécifier de valeur en entrée.

Soit, par exemple, l'en-tête de procédure suivant :

```
procedure ExtraitInfos(out Info: UnTypeEnreg);
```

Quand vous appelez *ExtraitInfos*, vous devez lui transmettre une variable de type *UnTypeEnreg* :

```
var MonEnreg: UnTypeEnreg;
 ⋮
ExtraitInfos(MonEnreg);
```

Mais vous n'utilisez pas *MonEnreg pour* transmettre des données à la procédure *ExtraitInfos* ; *MonEnreg* sert simplement de conteneur où *ExtraitInfos* stocke les informations qu'elle génère. L'appel de *ExtraitInfos* libère immédiatement la mémoire utilisée par *MonEnreg*, avant que le contrôle du programme ne passe à la procédure.

Les paramètres **Out** sont fréquemment utilisés avec les modèles d'objets distribués comme COM et CORBA. De plus, vous devez utiliser des paramètres **out** pour transmettre une variable non initialisée à une routine.

Paramètres sans type

Il est possible d'omettre la spécification de type quand vous déclarez des paramètres **var**, **const** ou **out**. Les paramètres valeur doivent être typés. Par exemple :

```
procedure FaitQuelquechose(const C);
```

déclare une procédure appelée *FaitQuelquechose* qui accepte un paramètre de type quelconque. Lorsque vous appelez une telle routine, vous ne pouvez pas lui passer un nombre ou une constante numérique non typée.

Dans le corps d'une procédure ou d'une fonction, les paramètres sans type sont incompatibles avec tous les types. Pour agir sur un paramètre sans type, vous devez le transtyper. En général, le compilateur ne peut vérifier si les opérations effectuées sur les paramètres sans type sont légales.

L'exemple suivant utilise des paramètres sans type dans une fonction appelée *Egal* qui compare le nombre spécifié d'octets de deux variables quelconques.

```
function Egal(var Source, Dest; Taille: Integer): Boolean;
type
  TOctets = array[0..MaxInt - 1] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Taille) and (TOctets(Dest)[N] = TOctets(Source)[N]) do
    Inc(N);
  Egal := N = Taille;
end;
```

Etant donné les déclarations suivantes :

```
type
  TVecteur = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVecteur;
  N: Integer;
  P: TPoint;
```

vous pouvez faire les appels suivants de la fonction *Egal* :

```
Egal(Vec1, Vec2, SizeOf(TVecteur))       // compare Vec1 et Vec2
Egal(Vec1, Vec2, SizeOf(Integer) * N)    // compare les N premiers éléments de Vec1 et Vec2
Egal(Vec1[1], Vec1[6], SizeOf(Integer) * 5) // compare les 5 premiers éléments aux 5
                                             // derniers éléments de Vec1
Egal(Vec1[1], P, 4)                          // compare Vec1[1] à P.X et Vec1[2] à P.Y
```

Paramètres chaîne

Lorsque vous déclarez des routines qui acceptent des paramètres chaîne courte, vous ne pouvez pas inclure des spécificateurs de longueur dans les déclarations de paramètres.

En effet, la déclaration

```
procedure Check(S: string[20]);   // erreur de syntaxe
```

génère une erreur de compilation. Mais

```
type TString20 = string[20];
procedure Check(S: TString20);
```

est correct. L'identificateur spécial *OpenString* peut être utilisé pour déclarer des routines qui prennent des paramètres chaîne courte de longueur variable :

procedure Check(S: OpenString);

Lorsque les directives de compilation **{$H–}** et **{$P+}** sont toutes deux effectives, le mot réservé **string** est équivalent à *OpenString* dans les déclarations de paramètres.

Les chaînes courtes, *OpenString*, **$H** et **$P** sont gérés uniquement dans un souci de compatibilité ascendante. Dans du nouveau code, vous pouvez ignorer ces considérations en utilisant des chaînes longues.

Paramètres tableau

Quand vous déclarez des routines utilisant des paramètres tableau, vous ne pouvez pas introduire de spécificateur de type d'indice dans la déclaration du paramètre. De ce fait, la déclaration suivante :

```
procedure Tri(A: array[1..10] of Integer);   // erreur de syntaxe
```

provoque une erreur de compilation. Mais

```
type TChiffres = array[1..10] of Integer;
procedure Tri(A: TChiffres);
```

est légal. Néanmoins, pour la plupart des cas, *les paramètres tableau ouvert* constituent une meilleure solution.

Paramètres tableau ouvert

Les paramètres tableau ouvert permettent de transmettre des tableaux de tailles différentes à la même routine. Pour définir une routine ayant un paramètre tableau ouvert, utilisez la syntaxe `array of` *type* (au lieu de `array[`*X*`..`*Y*`] of` *type*) dans la déclaration du paramètre. Par exemple :

```
function Trouver(A: array of Char): Integer;
```

déclare une fonction appelée *Trouver* qui prend comme paramètre un tableau de caractères de taille quelconque et renvoie un entier.

Remarque La syntaxe des paramètres tableau ouvert ressemble à celle des types de tableau dynamique mais elle ne signifie pas la même chose. L'exemple précédent crée une fonction qui accepte comme paramètre tout tableau d'éléments *Char* donc les tableaux dynamiques, mais uniquement eux.

Pour déclarer des paramètres qui doivent être des tableaux dynamiques, vous devez spécifier un identificateur de type :

```
type TTableauDynamiqueCaracteres = array of Char;
function Trouver(A: TTableauDynamiqueCaracteres): Integer;
```

Pour davantage d'informations sur les tableaux dynamiques, voir "Tableaux dynamiques" à la page 5-20.

Dans le corps d'une routine, les paramètres tableau ouvert sont régis par les règles suivantes

- Ce sont toujours des tableaux d'indice de base zéro. Le premier élément est 0, le second élément est 1, etc. Les fonctions standard *Low* et *High* renvoient, respectivement, 0 et *Length*−1. La fonction *SizeOf* renvoie la taille du tableau réellement transmis à la routine.
- On ne peut y accéder qu'élément par élément : l'affectation de la totalité d'un paramètre tableau ouvert est illégale.
- Ils ne peuvent être transmis à d'autres routines que comme paramètres tableau ouvert ou comme paramètres **var** sans type. Ils ne peuvent être transmis à *SetLength.*
- Au lieu d'un tableau, vous pouvez transmettre une variable du type de base du paramètre tableau ouvert. Elle est traitée comme un tableau de longueur 1.

Quand vous transmettez un tableau comme paramètre tableau ouvert par valeur, le compilateur crée une copie locale du tableau à l'intérieur du cadre de pile de la routine. Attention : vous ne devez pas provoquer de débordement du cadre de pile en transmettant de gros tableaux.

Les exemples suivants utilisent des paramètres tableau ouvert pour définir une procédure *Effacer* qui affecte la valeur zéro à chaque élément d'un tableau de réels et une fonction *Somme* qui calcule la somme de tous les éléments d'un tableau de réels.

```
procedure Effacer(var A: array of Real);
var
  I: Integer;
begin
  for I := 0 to High(A) do A[I] := 0;
end;

function Somme(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Somme := S;
end;
```

Quand vous appelez des routines qui utilisent des paramètres tableau ouvert, vous pouvez leur transmettre des constructeurs de tableau ouvert. Voir "Constructeurs de tableaux ouverts" à la page 6-20.

Paramètres tableau ouvert variant

Les paramètres tableau ouvert variant permettent de transmettre un tableau d'expressions de types différents à une seule routine. Pour définir une routine utilisant un paramètre tableau ouvert variant, spécifiez **array of const** comme type du paramètre. Ainsi :

```
procedure FaireQuelquechose(A: array of const);
```

déclare une procédure appelée *FaireQuelquechose* qui peut agir sur des tableaux de données hétérogènes.

La construction **array of const** est équivalente à **array of** *TVarRec*. *TVarRec*, déclaré dans l'unité *System*, représente un enregistrement avec une partie variable qui peut contenir des valeurs de type entier, booléen, caractère, réel, chaîne, pointeur, classe, référence de classe, interface et variant. *Le champ VType de TVarRec* indique le type de chaque élément du tableau. Certains types sont transmis comme pointeur et non comme valeur ; en particulier les chaînes longues sont transmises comme *Pointer* et doivent être transtypées en **string**. Pour davantage d'informations, voir l'aide en ligne sur *TVarRec*.

L'exemple suivant utilise un paramètre tableau ouvert variant dans une fonction qui crée une représentation sous forme de chaîne de chaque élément transmis et concatène le résultat dans une seule chaîne. Les routines de manipulation de chaînes utilisées dans cette function sont définies dans l'unité *SysUtils*.

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:    Result := Result + IntToStr(VInteger);
        vtBoolean:    Result := Result + BoolChars[VBoolean];
        vtChar:       Result := Result + VChar;
        vtExtended:   Result := Result + FloatToStr(VExtended^);
        vtString:     Result := Result + VString^;
        vtPChar:      Result := Result + VPChar;
        vtObject:     Result := Result + VObject.ClassName;
        vtClass:      Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:   Result := Result + CurrToStr(VCurrency^);
        vtVariant:    Result := Result + string(VVariant^);
        vtInt64:      Result := Result + IntToStr(VInt64^);
    end;
end;
```

Il est possible d'appeler cette fonction en utilisant un constructeur de tableau ouvert (voir “Constructeurs de tableaux ouverts” à la page 6-20). Par exemple,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

renvoie la chaîne “test100 T3.14159TForm”.

Paramètres par défaut

Il est possible de définir la valeur par défaut de paramètres dans l'en-tête d'une procédure ou d'une fonction. Les valeurs par défaut sont autorisées uniquement pour les paramètres **const** typés et les paramètres valeur. Pour spécifier une valeur par défaut, terminez la déclaration du paramètre par le symbole **=** suivi d'une expression constante compatible pour l'affectation avec le type du paramètre.

Par exemple, étant donné la déclaration suivante :

```
procedure RempliTableau(A: array of Integer; Valeur: Integer = 0);
```

les appels de procédure suivants sont équivalents :

```
RempliTableau(MonTableau);
RempliTableau(MonTableau, 0);
```

Une déclaration de paramètres multiples ne peut spécifier de valeur par défaut. Ainsi, la déclaration :

```
function MaFonction(X: Real = 3.5: Y: Real = 3.5): Real;
```

est légale, alors que la déclaration suivante :

```
function MaFonction(X, Y: Real = 3.5): Real;   // erreur de syntaxe
```

ne l'est pas.

Les paramètres ayant une valeur par défaut doivent se trouver à la fin de la liste des paramètres. C'est-à-dire que tous les paramètres suivant un paramètre ayant une valeur par défaut doivent également avoir une valeur par défaut. La déclaration suivante est donc illégale :

```
procedure MaProcedure(I: Integer = 1; S: string);   // erreur de syntaxe
```

Les valeurs par défaut spécifiées dans un type procédure ont la priorité sur celles spécifiées dans la routine réelle. Ainsi, étant donné les déclarations suivantes :

```
type TRedim = function(X: Real; Y: Real = 1.0): Real;
function Redim(X: Real; Y: Real = 2.0): Real;
var
  F: TRedim;
  N: Real;
```

les instructions :

```
F := Redim;
F(N);
```

provoquent le transfert des valeurs (*N*, 1.0) à *Redim*.

Les paramètres par défaut sont limités aux valeurs pouvant être spécifiées par une expression constante. (Voir "Expressions constantes" à la page 5-45.) De ce fait, les paramètres de type tableau dynamique, procédure, classe, référence de classe ou interface ne peuvent avoir que **nil** comme valeur par défaut. Les paramètres de type enregistrement, variant, fichier, tableau statique ou objet ne peuvent pas avoir du tout de valeur par défaut.

Pour davantage d'informations sur l'appel de routines avec des valeurs de paramètre par défaut, voir "Appel de procédures et de fonctions" à la page 6-19.

Paramètres par défaut et routines redéfinies

Si vous utilisez des valeurs par défaut de paramètre dans des routines redéfinies, évitez les signatures de paramètre ambiguës. Soit par exemple, le code suivant :

```
procedure Confus(I: Integer); overload;
  ⋮
procedure Confus(I: Integer; J: Integer = 0); overload;
  ⋮
Confus(X);    //  Quelle procédure est appelée?
```

En fait, aucune des deux procédures n'est appelée : ce code génère en effet une erreur de compilation.

Paramètres par défaut dans les déclarations forward et interface

Si une routine a une déclaration **forward** ou apparaît dans la section interface d'une unité, vous ne pouvez spécifier la valeur par défaut des paramètres que dans la déclaration **forward** ou dans la déclaration d'interface. Dans ce cas, les valeurs par défaut peuvent être omises dans la déclaration de définition (l'implémentation), mais si la déclaration de définition comprend des valeurs par défaut, elles doivent correspondre exactement avec celles de la déclaration **forward** ou de la déclaration d'interface

Appel de procédures et de fonctions

Quand vous appelez une procédure ou une fonction, le contrôle du programme passe du point d'appel au corps de la routine. Vous pouvez faire l'appel en utilisant le nom déclaré de la routine (avec ou sans qualificateur) ou en utilisant une variable procédure pointant sur la routine. Dans les deux cas, si la routine est déclarée avec des paramètres, votre appel doit transmettre des paramètres qui correspondent en ordre et en type à ceux spécifiés dans la liste des paramètres de la routine. Les paramètres transmis à la routine sont appelés *paramètres réels* alors que les paramètres de déclaration de la routine sont appelés *paramètres formels*.

Dans l'appel d'une routine, n'oubliez pas que :

- Les expressions utilisées pour transmettre des paramètres **const typés** et des paramètres valeur doivent être compatibles pour l'affectation avec les paramètres formels correspondants.
- Les expressions utilisées pour transmettre des paramètres **var** ou **out** doivent être de même type que les paramètres formels correspondants, sauf si le paramètre formel est sans type.
- Seules des expressions pouvant apparaître à gauche d'une affectation peuvent s'utiliser pour transmettre des paramètres **var** et **out**.

- Si les paramètres formels d'une routine sont sans type, les nombres et les constantes vraies avec des valeurs numériques ne peuvent pas être utilisés en tant que paramètres réels.

Quand vous appelez une routine qui utilise des valeurs de paramètre par défaut, tous les paramètres réels suivant le premier pour lequel vous utilisez la valeur par défaut doivent également utiliser leur valeur par défaut. Ainsi un appel de la forme `UneFonction(,,X)` est illégal.

Vous pouvez omettre les parenthèses lors de la transmission de tous les paramètres par défaut, et seulement des paramètres par défaut, à une routine. Par exemple, avec la procédure suivante

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

les appels suivants sont équivalents.

```
DoSomething();
DoSomething;
```

Constructeurs de tableaux ouverts

Les constructeurs de tableaux ouverts permettent de construire directement des tableaux dans l'appel de la procédure ou de la fonction. Ils ne peuvent être transmis que comme paramètres tableau ouvert ou comme paramètres tableau ouvert variant.

Un constructeur de tableau ouvert, comme un constructeur d'ensemble, est une série, placée entre crochets, d'expressions séparées par des virgules. Par exemple, étant donné les déclarations suivantes :

```
var I, J: Integer;
procedure Ajouter(A: array of Integer);
```

vous pouvez appeler la procédure *Ajouter* avec l'instruction :

```
Ajouter([5, 7, I, I + J]);
```

C'est l'équivalent du code suivant :

```
var Temp: array[0..3] of Integer;
 ⋮
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Ajouter(Temp);
```

Les constructeurs de tableau ouvert ne peuvent être transmis qu'à des paramètres valeur ou **const**. Les expressions du constructeur doivent être compatibles pour l'affectation avec le type de base du paramètre tableau. Dans le cas d'un paramètre tableau ouvert variant, les expressions peuvent être de types différents.

Chapitre

7

Classes et objets

Une *classe* (un *type classe)* définit une structure composée de *champs*, de *méthodes* et de *propriétés*. Les instances d'un type classe sont appelées des *objets*. Les champs, méthodes et propriétés d'une classe sont appelés ses *composants* ou ses *membres*.

- Un champ est essentiellement une variable faisant partie d'un objet. Comme les champs d'un enregistrement, un champ de classe représente des éléments de données qui existent dans chaque instance de la classe.
- Une méthode est une procédure ou une fonction associée à la classe. La plupart des méthodes portent sur des objets, c'est-à-dire sur des instances d'une classe. Certaines méthodes, appelées *méthodes de classe*, portent sur les types classe même.
- Une propriété est une interface avec les données associées à un objet (souvent stockées dans un champ). Les propriétés ont des *spécificateurs d'accès* qui déterminent comment leurs données sont lues et modifiées. Pour le reste d'un programme (hors de l'objet même), une propriété apparaît à bien des points de vue comme un champ.

Les objets sont des blocs de mémoire alloués dynamiquement dont la structure est déterminée par leur type de classe. Chaque objet détient une copie unique de chaque champ défini dans la classe. Par contre, toutes les instances d'une classe partagent les mêmes méthodes. Les objets sont créés et détruits par des méthodes spéciales appelées *constructeurs* et *destructeurs*.

Une variable de type classe est en fait un pointeur qui référence un objet. Plusieurs variables peuvent donc désigner le même objet. Comme les autres pointeurs, les variables de type classe peuvent contenir la valeur **nil**. Cependant, il n'est pas nécessaire de déréférencer explicitement une variable de type classe pour accéder à l'objet qu'elle désigne. Par exemple, `UnObjet.Taille := 100` affecte la valeur 100 à la propriété *Taille* de l'objet référencé, *UnObjet* ; vous ne *devez pas* l'écrire sous la forme `UnObjet^.Taille := 100`.

Types classe

Un type classe doit être déclaré et nommé avant de pouvoir être instancié. Il n'est donc pas possible de définir un type classe dans une déclaration de variable. Déclarez les classes uniquement dans la portée la plus large d'un programme ou d'une unité, mais pas dans une déclaration de procédure ou de fonction.

La déclaration d'un type classe a la forme suivante :

```
type nomClasse = class (classeAncêtre)
  listeMembre
 end;
```

où *nomClasse* est un identificateur valide, (*classeAncêtre*) est facultatif et *listeMembre* déclare les membres (les champs, méthodes et propriétés) de la classe. Si vous omettez (*classeAncêtre*), la nouvelle classe hérite directement de la classe prédéfinie *TObject*. Si vous précisez (*classeAncêtre*) en laissant vide *listeMembre*, vous pouvez omettre le `end` final. Une déclaration de type classe peut également contenir une liste des *interfaces* implémentées par la classe ; voir "Implémentation des interfaces" à la page 10-5.

Les méthodes apparaissent dans une déclaration de classe sous la forme d'en-tête de fonction ou de procédure sans le corps. La déclaration de définition de chaque méthode est faite ailleurs dans le programme.

Voici par exemple, la déclaration de la classe *TMemoryStream* de l'unité *Classes*.

```
type
TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity: Longint;
    procedure SetCapacity(NewCapacity: Longint);
  protected
    function Realloc(var NewCapacity: Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint; override;
  end;
```

TMemoryStream descend de *TStream* (dans l'unité *Classes*), et hérite de la plupart de ses membres. Mais elle définit, ou surcharge, plusieurs propriétés et méthodes, y compris la méthode destructeur, *Destroy*. Son constructeur, *Create*, est hérité sans modification de *TObject* et n'est donc pas redéclaré. Chaque membre est déclaré comme *private*, *protected* ou *public* (cette classe n'a pas de membre *published*) ; pour des informations sur ces termes, voir "Visibilité des membres de classes" à la page 7-4.

Etant donné cette déclaration, vous pouvez créer une instance de *TMemoryStream* comme ceci :

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

Héritage et portée

Quand vous déclarez une classe, vous pouvez spécifier son ancêtre immédiat. Par exemple :

```
type TUnControl = class(TControl);
```

déclare une classe appelée *TUnControle* qui descend (dérive) de *TControl*. Un type classe hérite automatiquement de tous les membres de son ancêtre immédiat. Chaque classe peut déclarer de nouveaux membres et redéfinir les membres hérités. Par contre, une classe ne peut supprimer des membres définis dans son ancêtre. Ainsi *TUnControle* contient tous les membres définis dans *TControl* et dans chacun des ancêtres de *TControl*.

La portée de l'identificateur d'un membre commence à l'endroit où le membre est déclaré et se poursuit jusqu'à la fin de la déclaration de la classe et s'étend à tous les descendants de la classe et les blocs de toutes les méthodes définies dans la classe et ses descendants.

TObject et TClass

La classe *TObject*, déclarée dans l'unité *System*, est l'ancêtre ultime de toutes les autres classes. *TObject* définit seulement quelques méthodes, dont un constructeur et un destructeur de base. Outre la classe *TObject*, l'unité *System* déclare le type référence de classe *TClass* :

```
TClass = class of TObject;
```

Pour davantage d'informations sur *TObject*, voir l'aide en ligne. Pour davantage d'informations sur les types référence de classe, voir “Références de classe” à la page 7-25.

Quand la déclaration d'un type classe ne spécifie pas d'ancêtre, la classe hérite directement de *TObject*. Donc :

```
type TMaClasse = class
 ⋮
end;
```

est équivalent à :

```
type TMaClasse = class(TObject)
 ⋮
end;
```

Cette dernière forme est recommandée dans un souci de lisibilité.

Compatibilité des types classe

Un type classe est compatible pour l'affectation avec ses ancêtres. Une variable d'un type classe peut donc référencer une instance de tout type descendant. Par exemple, étant donné la déclaration :

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure);
  TCarre = class(TRectangle);
var
  Fig: TFigure;
```

il est possible d'affecter à la variable *Fig* des valeurs de type *TFigure*, *TRectangle* *TCarre*.

Types objet

Comme alternative aux types classe, vous pouvez déclarer *des types objet* en utilisant la syntaxe :

```
type nomTypeObjet = object (typeObjetAncêtre)
  listeMembre
end;
```

où *nomTypeObjet* est un identificateur valide, (*typeObjetAncêtre*) est facultatif et *listeMembre* déclare les champs, méthodes et propriétés. Si (*typeObjetAncêtre*) n'est pas spécifié, le nouveau type n'a pas d'ancêtre. Les types objet ne peuvent pas avoir de membres publiés.

Comme les types objet ne descendent pas de *TObject*, ils ne disposent pas de constructeurs, de destructeurs ou d'autres méthodes prédéfinies. Vous pouvez créer des instances d'un type objet en utilisant la procédure *New* et les détruire en utilisant la procédure *Dispose*. Vous pouvez aussi déclarer tout simplement des variables d'un type objet comme pour un enregistrement.

Les types objet sont uniquement proposés dans un souci de compatibilité ascendante. Leur utilisation n'est pas recommandée.

Visibilité des membres de classes

Chaque membre d'une classe a un attribut appelé *visibilité*, indiqué par l'un des mots réservés suivants : **private**, **protected**, **public**, **published** ou **automated**. Par exemple :

```
published property Couleur: TColor read LitCouleur write EcritCouleur;
```

déclare une propriété publiée appelée *Couleur*. La visibilité détermine où et comment il est possible d'accéder à un membre : **private** (privée) représente l'accès minimum, **protected** (protégée) représente un niveau intermédiaire d'accès, **public** (publique), **published** (publiée) et **automated** (automatisée) représentant l'accès le plus large.

Si la déclaration d'un membre ne spécifie pas sa visibilité, le membre a la même visibilité que celui qui le précède. Les membres au début de la déclaration d'une

classe dont la visibilité n'est pas spécifiée sont par défaut publiés si la classe a été compilée dans l'état **{$M+}** ou si elle dérive d'une classe compilée à l'état **{$M+}** ; sinon ces membres sont publics.

Dans un souci de lisibilité, il est préférable d'organiser la déclaration d'une classe en fonction de la visibilité, en plaçant tous les membres privés ensemble, suivis de tous les membres protégés, etc. De cette manière, chaque mot réservé spécifiant la visibilité apparaît au maximum une fois et marque le début d'une nouvelle "section" de la déclaration. Une déclaration de classe standard doit donc avoir la forme :

```
type
  TMaClasse = class(TControl)
  private
   ⋮ { déclarations privées }
  protected
   ⋮ { déclarations protégées }
  public
   ⋮ { déclarations publiques }
  published
   ⋮ { déclarations publiées }
  end;
```

Vous pouvez augmenter la visibilité d'un membre dans une classe dérivée en le redéclarant, mais il n'est pas possible de réduire sa visibilité. Par exemple, une propriété protégée peut être rendue publique dans un descendant mais pas privée. De plus, les membres publiés ne peuvent devenir publics dans une classe dérivée. Pour davantage d'informations, voir "Surcharge et redéfinition de propriétés" à la page 7-24.

Membres privés, protégés et publics

Un membre *privé* est invisible hors de l'unité ou du programme dans lequel la classe est déclarée. En d'autres termes, une méthode privée ne peut être appelée depuis un autre module, et une propriété ou un champ privé ne peut être lu ou écrit depuis un autre module. En plaçant les déclarations de classes associées dans un même module, vous pouvez donner aux classes un accès à ces membres sans les rendre plus largement accessibles.

Un membre *protégé* est visible partout dans le module où la classe est déclarée et dans toute classe descendante, indépendamment du module où la classe descendante est définie. En d'autres termes, une méthode protégée peut être appelée et un champ ou une propriété lu ou écrit, dans la définition de toute méthode appartenant à une classe qui dérive de celle où le membre protégé est déclaré. Généralement, les membres qui ne servent qu'à l'implémentation des classes dérivées sont protégés.

Un membre *public* est visible partout où la classe peut être référencée.

Membres publiés

Les membres publiés ont la même visibilité que les membres publics. La différence est que des *informations de type à l'exécution* (RTTI) sont générées pour les membres publiés. Ces informations permettent à une application d'interroger

dynamiquement les champs et les propriétés d'un objet et de localiser ses méthodes. Les RTTI sont utilisés pour accéder aux valeurs des propriétés lors de la lecture ou de l'enregistrement des fichiers fiche, pour afficher les propriétés dans l'inspecteur de propriété et pour associer des méthodes spécifiques (appelées *gestionnaires d'événements*) à des propriétés spécifiques (appelées *événements*).

Les propriétés publiées sont limitées à certains types de données. Les types scalaires, chaîne, classe, interface et pointeur de méthode peuvent être publiés. Tout comme les types ensemble, dans la mesure où les bornes inférieure et supérieure du type de base ont des valeurs scalaires comprises entre 0 et 31. En d'autres termes, l'ensemble doit tenir dans un octet, un mot ou un double mot. Tous les types réels, à l'exception de *Real48, peuvent être publiés*. Les propriétés d'un type tableau (différentes des *propriétés tableau*, traitées ci-dessous) ne peuvent être publiées.

Ceratines propriétés, bien que publiables, ne sont pas totalement prises en charge par le système de flux. Il s'agit des propriétés de types enregistrement, des propriétés tableau de tous les types publiables (voir "Propriétés tableau" à la page 7-21), et des propriétés des types énumérés qui incluent des valeurs anonymes (voir "Types énumérés dont les rangs sont attribués explicitement" à la page 5-8). Si vous publiez une telle propriété, l'inspecteur d'objets ne l'affichera pas correctement, et la valeur de la propriété ne sera pas conservée quand les objets seront enregistrés sur disque.

Toutes les méthodes sont publiables, mais une classe ne peut publier plusieurs méthodes surchargées portant le même nom. Les champs ne peuvent être publiés que s'ils sont de type classe ou interface.

Une classe ne peut avoir de membres publiés que si elle est compilée avec l'état **{$M+}** ou si elle descend d'une classe compilée avec l'état **{$M+}**. La plupart des classes contenant des membres publiés dérivent de *TPersistent* qui est compilée avec l'état **{$M+}** il est donc rarement nécessaire d'utiliser la directive **$M**.

Membres automatisés

Les *membres automatisés* ont la même visibilité que les membres publics. La différence est que des *informations de type Automation* (requises pour les serveurs Automation) sont générés pour les membres automatisés. Les membres automatisés n'apparaissent généralement que dans les classes Windows et ne sont pas conseillés en programmation Linux. Le mot réservé **automated** n'est conservé que dans un souci de compatibilité ascendante.La classe *TAutoObject* de l'unité *ComObj* ne doit pas utiliser **automated**.

Les restrictions suivantes s'appliquent aux méthodes et propriétés déclarées comme automatisées :

- Le type de toutes les propriétés, des paramètres de propriété tableau, des paramètres de méthode et des résultats de fonction doivent être automatisables. Les types automatisables sont *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *WordBool* et tous les types interface.

- Les déclarations de méthode doivent utiliser la convention d'appel par défaut **register**. Les méthodes peuvent être virtuelles mais pas dynamiques.
- Les déclarations de propriété peuvent comporter des spécificateurs d'accès (**read** ou **write**) mais les autres spécificateurs (**index**, **stored**, **default** ou **nodefault**) sont interdits. Les spécificateurs d'accès doivent indiquer un identificateur de méthode utilisant la convention d'appel par défaut **register** ; les identificateurs de champ ne sont pas autorisés.
- Les déclarations de propriétés doivent spécifier un type. Il n'est pas permis de redéfinir les propriétés.

La déclaration d'une méthode ou d'une propriété automatisée peut inclure la directive **dispid**. La spécification d'un numéro d'identification déjà utilisé dans une directive **dispid** provoque une erreur.

Sous Windows, cette directive doit être suivie d'une constante entière qui spécifie un numéro d'identification de répartition Automation pour le membre. Sinon, le compilateur assigne automatiquement au membre un numéro d'identification de répartition plus grand que le plus grand numéro d'identification de répartition utilisé par les méthodes et les propriétés de la classe et de ses ancêtres. Pour plus d'informations sur l'Automation (sous Windows uniquement), voir “Objets automation (Windows seulement)” à la page 10-11.

Déclarations avancées et classes mutuellement dépendantes

Si la déclaration d'un type classe se termine par le mot **class** et un point-virgule, c'est-à-dire avec la forme :

```
type nomClasse = class;
```

sans ancêtre, ni liste des membres de classe après le mot **class** : c'est une *déclaration avancée*. Une déclaration avancée doit être complétée par une *déclaration de définition* de la même classe, dans la même section de déclaration de classe. En d'autres termes, entre une déclaration avancée et sa déclaration de définition, il ne peut y avoir que d'autres déclarations de type.

Les déclarations avancées permettent de définir des classes mutuellement dépendantes. Par exemple :

```
type
  TFigure = class;  // déclaration avancée
  TDessin = class
    Figure: TFigure;
     ⋮
  end;

  TFigure = class  // déclaration de définition
    Dessin: TDessin;
     ⋮
  end;
```

Ne confondez pas les déclarations avancées avec les déclarations complètes de type qui dérivent de *TObject* sans déclarer de membres de classe :

```
type
  TClasseUn = class;              // c'est une déclaration avancée

  TClasseDeux = class             // c'est une déclaration de classe complète
    end;                          //

  TClasseTrois = class(TObject);  // c'est une déclaration de classe complète
```

Champs

Un champ est semblable à une variable appartenant à un objet. Les champs peuvent être de type quelconque, y compris de type classe (les champs peuvent donc contenir des références d'objet). Les champs sont généralement privés.

Pour définir un membre champ dans une classe, déclarez simplement le champ comme pour une variable. Toutes les déclarations de champ doivent être placées avant les déclarations de propriétés ou de méthodes. Par exemple, la déclaration suivante crée une classe appelée *TNombre* dont le seul membre, en dehors des méthodes héritées de *TObject*, est un champ entier appelé *Int* :

```
type TNombre = class
  Int: Integer;
end;
```

Les champs sont liés statiquement, c'est-à-dire que les références les désignant sont fixées à la compilation. Pour comprendre ce que cela signifie, examinez le code suivant :

```
type
  TAncetre = class
    Valeur: Integer;
  end;

  TDescendant = class(TAncetre)
    Valeur: string;  // masque le champ Valeur hérité
  end;

var
  MonObjet: TAncetre;

begin
  MonObjet := TDescendant.Create;
  MonObjet.Valeur := 'Hello!';              // erreur
  TDescendant(MonObjet).Valeur := 'Hello!';  // correct !
end;
```

Même si *MonObjet* contient une instance de *TDescendant*, il est déclaré comme un *TAncetre*. Le compilateur interprète donc *MonObjet.Valeur* comme une référence au champ (entier) déclaré dans *TAncetre*. Cependant, les deux champs existent dans l'objet *TDescendant* ; la valeur *Valeur* héritée est masquée par la nouvelle, mais il est possible d'y accéder en utilisant un transtypage.

Méthodes

Une méthode est une procédure ou une fonction associée à une classe. Un appel de méthode spécifie l'objet (ou la classe si c'est une méthode de classe) sur lequel la méthode agit. Par exemple :

```
UnObjet.Free
```

appelle la méthode *Free* de *UnObjet*.

Déclarations et implémentations des méthodes

A l'intérieur d'une déclaration de classe, une méthode apparaît comme un en-tête de procédure ou de fonction qui fonctionne comme une déclaration avancée (**forward)**. Quelque part après la déclaration de classe, mais à l'intérieur du même module, chaque méthode doit être implémentée par une déclaration de définition. Si par exemple, la déclaration de *TMaClasse* contient une méthode appelée *FaireQuelquechose*:

```
type
  TMaClasse = class(TObject)
     ⋮
    procedure FaireQuelquechose;
     ⋮
  end;
```

La déclaration de définition de *FaireQuelquechose* doit se trouver après dans le même module :

```
procedure TMaClasse.FaireQuelquechose;
begin
 ⋮
end;
```

Si une classe peut être déclarée dans la section interface ou dans la section implémentation d'une unité, les déclarations de définition des méthodes d'une classe doivent se trouver dans la section implémentation.

Dans l'en-tête d'une déclaration de définition, le nom de la méthode est toujours qualifié par le nom de la classe à laquelle elle appartient. L'en-tête peut répéter la liste de paramètres de la déclaration de classe ; si c'est le cas, l'ordre, le type et le nom des paramètres doivent correspondre exactement, et, si la méthode est une fonction, cela doit également être le cas pour le type de valeur renvoyée.

Les déclarations des méthodes pauvent inclure des directivesspéciales qui ne sont pas utilisées par d'autres fonctions ou procédures. Les directives doivent apparaître uniquement dans la déclaration de classe et pas dans la déclaration de définition, et doivent toujours être dans l'ordre suivant :

reintroduce; **overload**; *liaison*; *convention d'appel*; **abstract**; *avertissement*

où *liaison* est **virtual**, **dynamic** ou **override**; *convention d'appel* est **register**, **pascal**, **cdecl**, **stdcall** ou **safecall**; et *avertissement* est **platform**, **deprecated** ou **library**.

Inherited

Le mot réservé **inherited** joue un rôle particulier dans l'implémentation de comportements polymorphiques. Il peut apparaître dans une définition de méthode avec ou sans identificateur à la suite.

Si **inherited** est suivi par du nom d'un membre, il représente un appel de méthode normal ou une référence à une propriété ou à un champ, sauf que la recherche du membre référencé commence dans l'ancêtre immédiat de la classe de la méthode. Ainsi, quand l'instruction :

```
inherited Create(...);
```

se produit dans la définition d'une méthode, elle appelle la méthode *Create héritée*.

Quand **inherited** est utilisé sans être suivi d'un identificateur, il désigne la méthode héritée portant le même nom que la méthode en cours. Dans ce cas, **inherited** ne prend pas de paramètre explicite, mais transmet à la méthode héritée les paramètres utilisés pour l'appel de la méthode en cours. Ainsi :

```
inherited;
```

apparaît fréquemment dans l'implémentation des constructeurs. Cette instruction appelle le constructeur hérité en lui transmettant les mêmes paramètres que ceux transmis au descendant.

Self

A l'intérieur de l'implémentation d'une méthode, l'identificateur *Self* désigne l'objet dans lequel la méthode est appelée. Voici, par exemple, l'implémentation de la méthode *Add de TCollection* dans l'unité *Classes*.

```
function TCollection.Add: TCollectionItem;
begin
  Result := FItemClass.Create(Self);
end;
```

La méthode *Add* appelle la méthode *Create* de la classe référencée par le champ *FItemClass* qui est toujours un descendant de *TCollectionItem*. *TCollectionItem.Create* ne prend qu'un seul paramètre de type *TCollection*, donc *Add* le transmet à l'objet instance de *TCollection* d'où *Add* est appelée. Cela est illustré par le code suivant :

```
var MaCollection: TCollection;
 ⋮
MaCollection.Add  // MaCollection est transmis à la méthode TCollectionItem.Create
```

Self est utilisé dans divers cas de figure. Par exemple, un identificateur de membre déclaré dans un type classe peut être redéclaré dans le bloc de l'une des méthodes de la classe. Dans ce cas, vous pouvez accéder à l'identificateur du membre d'origine en utilisant `Self`.*Identificateur*.

Pour davantage d'informations sur l'utilisation de *Self* dans les méthodes de classe, voir "Méthodes de classe" à la page 7-28.

Liaison de méthode

Les méthodes peuvent être *statiques* (c'est le cas par défaut), *virtuelles* ou *dynamiques*. Les méthodes virtuelles et dynamiques peuvent être *surchargées* et elles peuvent être *abstraites*. Ces éléments jouent un rôle quand une variable d'un type classe contient une valeur d'un type classe descendant. Ils déterminent quelle implémentation est activée lors de l'appel d'une méthode.

Méthodes statiques

Par défaut les méthodes sont statiques. Quand une méthode statique est appelée, le type déclaré (à la compilation) de la variable classe ou objet est utilisé dans l'appel de méthode pour déterminer l'implémentation à activer. Dans l'exemple suivant, les méthodes *Dessiner* sont statiques :

```
type
  TFigure = class
    procedure Dessiner;
  end;
  TRectangle = class(TFigure)
    procedure Dessiner;
  end;
```

Etant donné ces déclarations, le code suivant illustre l'effet de l'appel d'une méthode statique. Dans le second appel de *Figure.Dessiner*, la variable *Figure* désigne un objet de la classe *TRectangle*, mais l'appel utilise l'implémentation de *Dessiner de TFigure*, car le type déclaré de la variable *Figure* est *TFigure*.

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Dessiner;             // appelle TFigure.Dessiner
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Dessiner;             // appelle TFigure.Dessiner
  TRectangle(Figure).Dessiner; // appelle TRectangle.Dessiner
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Dessiner;          // appelle TRectangle.Dessiner
  Rectangle.Destroy;
end;
```

Méthodes virtuelles et dynamiques

Pour définir une méthode comme étant virtuelle ou dynamique, incluez les directives **virtual** ou **dynamic** dans la déclaration. Les méthodes dynamiques ou virtuelles, à la différence des méthodes statiques, peuvent être *surchargées* dans les classes dérivées. Quand une méthode surchargée est appelée, c'est le type réel (à l'exécution) du type de classe ou d'objet utilisé dans l'appel de la méthode, et non pas le type déclaré de la variable, qui détermine l'implémentation activée.

Pour surcharger une méthode, redéclarez-là avec la directive **override**. Une déclaration **override** doit correspondre à la déclaration de l'ancêtre dans l'ordre et le type des paramètres, ainsi que dans le type éventuel du résultat.

Dans l'exemple suivant, la méthode *Dessiner* déclarée dans *TFigure* est surchargée dans deux classes dérivées :

```
type
  TFigure = class
    procedure Dessiner; virtual;
  end;
  TRectangle = class(TFigure)
    procedure Dessiner; override;
  end;
  TEllipse = class(TFigure)
    procedure Dessiner; override;
  end;
```

Etant donné ces déclarations, le code suivant illustre le résultat de l'appel d'une méthode virtuelle via une variable dont le type réel change à l'exécution :

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Dessiner;  // appelle TRectangle.Dessiner
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Dessiner;  // appelle TEllipse.Dessiner
  Figure.Destroy;
end;
```

Seules les méthodes virtuelles et dynamiques peuvent être surchargées. Par contre, toutes les méthodes peuvent être *redéfinies* ; voir “Redéfinition de méthodes” à la page 7-13.

Comparaison des méthodes virtuelles et des méthodes dynamiques

D'un point de vue sémantique, les méthodes virtuelles et les méthodes dynamiques sont équivalentes. Elles ne diffèrent que dans l'implémentation de la répartition de l'appel de méthode à l'exécution. Les méthodes virtuelles sont optimisées pour la rapidité alors que les méthodes dynamiques permettent d'optimiser la taille du code.

En général, les méthodes virtuelles constituent la manière la plus efficace d'implémenter un comportement polymorphique. Les méthodes dynamiques sont utiles quand une classe de base déclare de nombreuses méthodes pouvant être surchargées qui sont héritées par de nombreuses classes dérivées d'une application mais rarement surchargées.

Surcharge ou masque

Si une déclaration de méthode spécifie le même identificateur de méthode et la même signature de paramètres qu'une méthode héritée sans spécifier la directive **override**, la nouvelle déclaration *masque* simplement la méthode héritée sans la

surcharger. Les deux méthodes existent alors dans la classe dérivée où le nom de méthode est lié statiquement. Par exemple :

```
type
  T1 = class(TObject)
    procedure Agir; virtual;
  end;
  T2 = class(T1)
    procedure Agir;  // Agir est redéclarée mais pas surchargée
  end;

var
  UnObjet: T1;
begin
  UnObjet := T2.Create;
  UnObjet.Agir;      // appelle T1.Agir
end;
```

Reintroduce

La directive **reintroduce** supprime les avertissements du compilateur informant qu'une méthode virtuelle précédemment déclarée est masquée. Par exemple :

```
procedure FaireQuelquechos; reintroduce;  // la classe ancêtre a aussi une méthode
FaireQuelquechose
```

Utilisez **reintroduce** quand vous voulez masquer une méthode virtuelle héritée avec une nouvelle méthode.

Méthodes abstraites

Une méthode abstraite est une méthode virtuelle ou dynamique n'ayant pas d'implémentation dans la classe où elle est déclarée. Son implémentation est déléguée à une classe dérivée. Les méthodes abstraites doivent être déclarées en spécifiant la directive **abstract** après **virtual** ou **dynamic**. Par exemple :

```
procedure FaireQuelquechose; virtual; abstract;
```

Vous ne pouvez appeler une méthode abstraite que dans une classe ou une instance de classe dans laquelle la méthode a été surchargée.

Redéfinition de méthodes

Une méthode peut être redéclarée en utilisant la directive **overload**. Dans ce cas, si la méthode redéclarée a une signature de paramètres différente de celle de son ancêtre, elle redéfinit la méthode héritée sans la cacher. L'appel de la méthode dans une classe dérivée active l'implémentation qui correspond aux paramètres utilisés dans l'appel.

Si vous redéfinissez une méthode virtuelle, utilisez la directive **reintroduce** quand vous la redéclarez dans les classes dérivées. Par exemple :

```
type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
```

```
  end;
  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
   ⋮
  UnObjet := T2.Create;
  UnObjet.Test('Hello!');  // appelle T2.Test
  UnObjet.Test(7);         // appelle T1.Test
```

A l'intérieur d'une classe, vous ne pouvez pas publier de multiples méthodes redéfinies avec le même nom. La maintenance des informations de type à l'exécution requiert un nom unique pour chaque membre publié.

```
type
  TSomeClass = class
  published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer  // erreur
     ⋮
```

Les méthodes qui servent de spécificateurs de lecture ou d'écriture de propriétés ne peuvent pas être redéfinies.

L'implémentation d'une méthode surchargée doit répéter la liste des paramètres spécifiée dans la liste de paramètres de la déclaration de classe. Pour davantage d'informations sur la redéfinition, voir "Redéfinition de procédures et de fonctions" à la page 6-8.

Constructeurs

Un constructeur est une méthode spéciale qui crée et initialise des instances d'objet. La déclaration d'un constructeur ressemble à celle d'une procédure en commençant par le mot **constructor**. Exemples :

```
constructor Create;
constructor Create(AOwner: TComponent);
```

Les constructeurs doivent utiliser la convention d'appel **register** par défaut. Bien que la déclaration ne spécifie pas de valeur renvoyée, un constructeur renvoie une référence à l'objet qu'il a créé ou qui est appelé.

Une classe peut avoir plusieurs constructeurs mais doit en avoir au moins un. L'habitude veut qu'on appelle le constructeur *Create*.

Pour créer un objet, appelez la méthode constructeur dans un type classe. Par exemple :

```
MonObjet := TMaClasse.Create;
```

Cette instruction alloue sur le tas le stockage pour le nouvel objet, initialise la valeur de tous les champs scalaires à zéro, affecte **nil** à tous les champs de type pointeur ou de type classe, et une chaîne vide à tous les champs chaîne. Les autres actions spécifiées dans l'implémentation du constructeur sont effectuées ensuite. Généralement, les objets sont initialisés en fonction des valeurs transmises comme paramètres au constructeur. Enfin, le constructeur renvoie une

référence à l'objet qui vient d'être créé et initialisé. Le type de valeur renvoyée est le même que celui du type classe spécifié dans l'appel du constructeur.

Si une exception est déclenchée lors de l'exécution d'un constructeur appelé dans une référence de classe, le destructeur *Destroy* est appelé automatiquement pour détruire l'objet inachevé.

Quand un constructeur est appelé en utilisant une référence d'objet (au lieu d'une référence de classe), il ne crée pas d'objet. Le constructeur agit à la place sur l'objet spécifié en n'exécutant que les instructions de l'implémentation du constructeur et renvoie ensuite une référence à l'objet. Un constructeur est généralement appelé dans une référence d'objet en conjonction avec le mot réservé **inherited** afin d'exécuter un constructeur hérité.

Voici un exemple de type classe et de son constructeur :

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ⋮
  end;

constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);  // Initialise les parties héritées
  Width := 65;  // Modifie les propriétés héritées
  Height := 65;
  FPen := TPen.Create;  // Initialise les nouveaux champs
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

Généralement, la première action d'un constructeur est d'appeler le constructeur hérité afin d'initialiser les champs hérités de l'objet. Le constructeur initialise ensuite les champs introduits dans la classe dérivée. Comme un constructeur efface toujours le stockage alloué à un nouvel objet, tous les champs contiennent au départ zéro (pour les types scalaires), **nil** (types pointeur et classe) chaîne vide (types chaîne) ou *Unassigned* (variants). Il n'est donc pas nécessaire que l'implémentation du constructeur initialise les champs sauf ceux devant contenir une valeur non nulle ou non vide.

Quand il est appelé via un identificateur de type classe, un constructeur déclaré comme **virtual** est équivalent à un constructeur statique. Par contre, quand ils sont combinés avec des types référence de classe, les constructeurs virtuels permettent une construction polymorphique des objets : c'est-à-dire la construction d'objets dont le type est inconnu à la compilation, voir "Références de classe" à la page 7-25.

Destructeurs

Un destructeur est une méthode spéciale qui détruit l'objet à l'endroit de son appel et libère sa mémoire. La déclaration d'un destructeur ressemble à celle d'une procédure mais elle commence par le mot **destructor**. Par exemple :

```
destructor Destroy;
destructor Destroy; override;
```

Les destructeurs doivent utiliser la convention d'appel **register** par défaut. Même si une classe peut avoir plusieurs destructeurs, il est conseillé que chaque classe surcharge la méthode *Destroy* héritée et ne déclare pas d'autres destructeurs.

Pour appeler un destructeur, il faut référencer une instance d'objet. Par exemple :

```
MonObjet.Destroy;
```

Lors de l'appel d'un destructeur, les actions spécifiées dans l'implémentation du destructeur sont d'abord exécutées. Généralement, cela consiste à détruire les objets incorporés et libérer les ressources allouées par l'objet. Ensuite, le stockage alloué à l'objet est libéré.

Voici un exemple d'implémentation d'un destructeur :

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

Généralement, la dernière action de l'implémentation d'un destructeur est l'appel du destructeur hérité afin de détruire les champs hérités de l'objet.

Quand une exception est déclenchée lors de la création d'un objet, *Destroy* est appelée automatiquement afin de libérer l'objet inachevé. Cela signifie que *Destroy* doit être capable de libérer des objets partiellement construits. Comme un constructeur commence par initialiser à zéro ou à des valeurs vides les champs d'un nouvel objet avant d'effectuer d'autres actions, les champs de type classe ou pointeur d'un objet partiellement construit ont toujours la valeur **nil**. Un destructeur doit donc tester les valeurs **nil** avant d'agir sur des champs de type classe ou pointeur. L'appel de la méthode *Free* (définie dans *TObject*) au lieu de *Destroy*, permet de tester facilement les valeurs **nil** avant de détruire un objet.

Méthodes de messages

Les méthodes de messages implémentent des réponses à des messages répartis dynamiquement. La syntaxe des méthodes de messages est supportée sur toutes les plates-formes. La VCL utilise des méthodes de messages pour répondre aux messages Windows. CLX n'utilise pas les méthodes de messages pour répondre aux événements système.

Un gestionnaire de messages est créé en incluant la directive **message** dans une déclaration de méthode, suivie d'une constante entière comprise entre 1 et 49151

qui spécifie le numéro d'identification du message. Pour les méthodes de messages des contrôles VCL, la constante entière peut être un des numéros d'identification de message Windows, avec les types d'enregistrements correspondants, dans l'unité *.Messages*. Un gestionnaire de messages doit être une procédure qui n'attend qu'un seul paramètre **var**.

Par exemple, sous Windows :

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ⋮
  end;
```

Par exemple, sous Linux ou pour la programmation multiplate-forme, vous gèreriez les messages comme ceci :

```
const
  ID_REFRESH = $0001;

type
  TTextBox = class(TCustomControl)
  private
    procedure Refresh(var Message: TMessageRecordType); message ID_REFRESH;
    ⋮
  end;
```

Un gestionnaire de messages doit être une procédure qui n'attend qu'un seul .

La déclaration d'une méthode de messages n'a pas besoin de spécifier la directive **override** pour surcharger une méthode de messages héritée. En fait, il n'est même pas nécessaire qu'elle porte le même nom de méthode ou qu'elle utilise les mêmes types de paramètres que la méthode qu'elle surcharge. C'est le numéro d'identification du message seul qui détermine les messages auxquels la méthode répond et la méthode qu'elle surcharge.

Implémentation des méthodes de messages

L'implémentation d'une méthode de messages peut appeler la méthode de message héritée, comme dans l'exemple suivant (pour Windows) :

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
    ProcessEnter
  else
    inherited;
end;
```

Sous Linux ou pour la programmation multiplate-forme, vous écririez le même exemple comme ceci :

```
procedure TTextBox.Refresh(var Message: TMessageRecordType);
begin
  if Chr(Message.Code) = #13 then
    ...
```

```
    else
      inherited;
  end;
```

L'instruction **inherited** recherche en arrière dans la hiérarchie des classes la première méthode de messages ayant le même numéro d'identification que la méthode en cours et lui transmet automatiquement l'enregistrement message. Si aucune classe ancêtre n'implémente de méthode de messages pour le numéro d'identification spécifié, **inherited** appelle la méthode *DefaultHandler* définie initialement dans *TObject*.

L'implémentation de *DefaultHandler* dans *TObject* rend le contrôle sans rien faire. En redéfinissant *DefaultHandler*, une classe peut implémenter sa propre gestion par défaut des messages. Sous Windows, la méthode *DefaultHandler* pour les contrôles VCL appelle la fonction Windows *DefWindowProc*.

Répartition des messages

Les gestionnaires de messages sont rarement appelés directement. Les messages sont répartis à un objet en utilisant la méthode *Dispatch* héritée de *TObject* :

```
procedure Dispatch(var Message);
```

Le paramètre *Message* transmis à *Dispatch* doit être un enregistrement dont la première entrée est un champ de type *Cardinal* contenant le numéro d'identification du message.

La méthode Dispatch recherche en arrière dans la hiérarchie des classes, en commençant par la classe de l'objet où elle est appelée, et appelle la première méthode de messages pour le numéro d'identification qui lui a été transmis. Si aucune méthode de messages ne correspond au numéro d'identification transmis, *Dispatch* appelle *DefaultHandler*.

Propriétés

Comme un champ, une propriété définit un attribut d'un objet. Mais alors qu'un champ n'est rien de plus qu'un emplacement de stockage dont le contenu peut être consulté et modifié, une propriété associe des actions spécifiques à la lecture et la modification de ses données. Les propriétés proposent un moyen de contrôler l'accès aux attributs d'un objet et permettent le calcul des attributs.

La déclaration d'une propriété spécifie son nom et son type et contient au moins un spécificateur d'accès. Une déclaration de propriété a la syntaxe suivante :

```
property nomPropriété[indices]: type index constanteEntière spécificateurs;
```

où

- *nomPropriété* est un identificateur valide.
- [*indices*] est facultatif, c'est une suite de déclarations de paramètre séparées par des point-virgules. Chaque déclaration de paramètre est de la forme $identificateur_1$ ... $identificateur_n$: *type*. Pour davantage d'informations, voir "Propriétés tableau" à la page 7-21.

- *type* doit être prédéfini ou un type de données déclaré précédemment. C'est-à-dire que les déclarations de propriétés comme `property Num: 0..9 ...` sont invalides.
- La clause `index` *constanteEntière* est facultative. Pour davantage d'informations, voir "Spécificateurs d'indice" à la page 7-22.
- *spécificateurs* est une suite contenant les spécificateurs : **read**, **write**, **stored**, **default** (ou **nodefault**) ou **implements**. Chaque déclaration de propriété doit comporter au moins un spécificateur **read** ou **write**. Pour des informations sur **implements**, voir "Implémentation des interfaces par délégation" à la page 10-7.)

Les propriétés sont définies par leur spécificateurs d'accès. A la différence des champs, les propriétés ne peuvent être transmises comme paramètre **var**. De même, il n'est pas possible d'appliquer l'opérateur @ à une propriété. En effet, une propriété n'existe pas nécessairement en mémoire. Elle peut, par exemple, avoir une méthode **read** qui obtient une valeur d'une base de données ou génère une valeur aléatoire.

Accès aux propriétés

Chaque propriété a un spécificateur **read**, un spécificateur **write** ou les deux. Ce sont les *spécificateurs d'accès*. Ils ont la forme suivante :

`read` *champOuMéthode*
`write` *champOuMéthode*

où *champOuMéthode* est le nom du champ ou de la méthode déclaré dans la même classe ou dans une classe ancêtre.

- Si *champOuMéthode* est déclaré dans la même classe, il doit apparaître avant la déclaration de la propriété. S'il est déclaré dans une classe ancêtre, il doit être visible dans le descendant ; ce ne peut donc pas être un champ ou une méthode privé d'une classe ancêtre déclarée dans une autre unité.
- Si *champOuMéthode* est un champ, il doit être de même type que la propriété.
- Si *champOuMéthode* est une méthode, elle ne peut être redéfinie. En outre, le méthodes d'accès pour une propriété publiée doivent utiliser la convention d'appel par défaut **register**.
- Dans un spécificateur **read**, si *champOuMéthode* est une méthode, ce doit être une fonction sans paramètre renvoyant une valeur de même type que celui de la propriété.
- Dans un spécificateur **write**, si *champOuMéthode* est une méthode, ce doit être une procédure prenant un seul paramètre valeur ou **const** du même type que la propriété.

Par exemple, étant donné les déclarations suivantes :

```
property Couleur: TColor read LitCouleur write EcritCouleur;
```

la méthode *LitCouleur* doit être déclarée comme suit :

```
function LitCouleur: TColor;
```

et la méthode *EcritCouleur* doit utiliser l'une des déclarations suivantes :

```
procedure EcritCouleur(Valeur: TColor);
procedure EcritCouleur(const Valeur: TColor);
```

Le nom du paramètre de *EcritCouleur* n'a pas besoin d'être *Valeur*.

Quand une propriété est référencée dans une expression, sa valeur est lue en utilisant le champ ou la méthode indiqué par le spécificateur **read**. Quand une propriété est référencée dans une instruction d'affectation, sa valeur est modifiée en utilisant le champ ou la méthode indiqué par le spécificateur **write**.

L'exemple suivant déclare une classe appelée *TCompas* ayant une propriété publiée *Direction*. La valeur de *Direction* est lue via le champ *FDirection* et écrite via la procédure *EcritDirection*.

```
type
  TDirection = 0..359;
  TCompas = class(TControl)
  private
    FDirection: TDirection;
    procedure EcritDirection(Valeur: TDirection);
  published
    property Direction: TDirection read FDirection write EcritDirection;
    ⋮
  end;
```

Etant donné cette déclaration, les instructions :

```
if Compas.Direction = 180 then VaAuSud;
Compas.Direction := 135;
```

correspondent à :

```
if Compas.FDirection = 180 then VaAuSud;
Compas.EcritDirection(135);
```

Dans la classe *TCompas*, aucune action n'est associée à la lecture de la propriété *Direction* ; l'opération **read** consiste simplement à récupérer la valeur stockée dans le champ *FDirection*. Par contre, l'affectation d'une valeur à la propriété *Direction* se transforme en un appel de la méthode *EcritDirection* qui, probablement, stocke la nouvelle valeur dans le champ *FDirection* mais effectue également d'autres actions. La méthode *EcritDirection* peut, par exemple, être implémentée de la manière suivante :

```
procedure TCompas.EcritDirection(Valeur: TDirection);
begin
  if FDirection <> Valeur then
  begin
    FDirection := Valeur;
    Repaint;  // actualise l'interface utilisateur afin de refléter la nouvelle valeur
  end;
end;
```

Une propriété dont la déclaration ne contient qu'un spécificateur **read** est une propriété *en lecture seule*, et la propriété dont la déclaration ne comporte que le spécificateur **write** est une propriété *en écriture seulement*. Affecter une valeur à une propriété en lecture seule ou utiliser une propriété en écriture seulement dans une expression sont des erreurs.

Propriétés tableau

Les propriétés tableau sont des propriétés indicées. Elles peuvent représenter les éléments d'une liste, les contrôles enfant d'un contrôle ou les pixels d'un bitmap.

La déclaration d'une propriété tableau inclut une liste de paramètres spécifiant le nom et le type des indices. Par exemple :

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

Le format d'une liste de paramètres indice est le même que celui de la liste de paramètres d'une procédure ou d'une fonction, à cette différence que la déclaration des paramètres est placée entre crochets au lieu de parenthèses. A la différence des tableaux qui ne peuvent utiliser que des indices de type scalaire, les propriétés tableau peuvent avoir des indices de type quelconque.

Pour les propriétés tableau, les spécificateurs d'accès doivent indiquer des méthodes et pas des champs. La méthode d'un spécificateur **read** doit être une fonction ayant les mêmes paramètres, en nombre, en ordre et en type, que ceux indiqués dans la liste de paramètres indice de la propriété et dont le type de résultat est identique au type de la propriété. La méthode indiquée par le spécificateur **write** doit être une procédure ayant les mêmes paramètres, en nombre, en ordre et en type, que ceux indiqués dans la liste de paramètres indice de la propriété plus un paramètre valeur ou **const** supplémentaire du même type que la propriété.

Par exemple, les méthodes d'accès des propriétés tableau déclarées dans l'exemple précédent peuvent se déclarer de la manière suivante :

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

Vous accédez à une propriété tableau en indiçant l'identificateur de la propriété.

Par exemple, les instructions :

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

correspondent à :

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

Sous Linux, vous utiliseriez un chemin du type '/usr/local/bin' au lieu de 'C:\DELPHI\BIN' dans l'exemple précédent.

La définition d'une propriété tableau peut être suivie de la directive **default**, auquel cas, la propriété tableau devient la propriété par défaut de la classe. Par exemple :

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    ⋮
  end;
```

Si une classe a une propriété par défaut, vous pouvez accéder à cette propriété en utilisant la forme abrégée *objet*[*indice*] équivalente à *objet*.*propriété*[*indice*]. Par exemple, étant donné les déclarations précédentes, `StringArray.Strings[7]` peut s'abréger en `StringArray[7]`. Une classe ne peut avoir qu'une seule propriété par défaut. Changer ou masquer la propriété par défaut dans les classes dérivées peut entraîner un comportement inattendu, car le compilateur détermine toujours la propriété par défaut d'un objet de manière statique.

Spécificateurs d'indice

Les spécificateurs d'indice permettent à plusieurs propriétés de partager les mêmes méthodes d'accès tout en représentant des valeurs différentes. Un spécificateur d'indice est constitué de la directive **index** suivie d'une constante entière comprise entre –2147483647 et 2147483647. Si une propriété a un spécificateur d'indice, ses spécificateurs **read** et **write** doivent indiquer des méthodes et pas des champs. Par exemple :

```
type
  TRectangle = class
  private
    FCoordonnees: array[0..3] of Longint;
    function LitCoordonnees(Indice: Integer): Longint;
    procedure EcritCoordonnees(Indice: Integer; Valeur: Longint);
  public
    property Gauche: Longint index 0 read LitCoordonnees write EcritCoordonnees;
    property Haut: Longint index 1 read LitCoordonnees write EcritCoordonnees;
    property Droite: Longint index 2 read LitCoordonnees write EcritCoordonnees;
    property Bas: Longint index 3 read LitCoordonnees write EcritCoordonnees;
    property Coordonnees [Indice: Integer]: Longint read LitCoordonnees write
EcritCoordonnees;
    ⋮
  end;
```

La méthode d'accès d'une propriété ayant un spécificateur d'accès doit prendre un paramètre supplémentaire de type *Integer*. Pour une fonction **read**, ce doit être le dernier paramètre ; pour une procédure **write,** ce doit être l'avant-dernier paramètre (avant le paramètre spécifiant la valeur de la propriété). Quand un programme accède à la propriété, la constante entière est automatiquement transmise à la méthode d'accès.

Etant donné les déclarations précédentes, si *Rectangle* est de type *TRectangle*, alors :

```
Rectangle.Droite := Rectangle.Gauche + 100;
```

correspond à :

```
Rectangle.EcritCoordonnees(2, Rectangle.LitCoordonnees(0) + 100);
```

Spécificateurs de stockage

Les directives facultatives **stored**, **default** et **nodefault** sont appelées des *spécificateurs de stockage*. Elles n'ont aucun effet sur l'exécution du programme mais contrôlent la manière dont les informations de type à l'exécution (RTTI) sont gérées. Plus précisément, les spécificateurs de stockage indiquent si on enregistre ou non la valeur des propriétés publiées dans les fichiers fiche (.DFM).

La directive **stored** doit être suivie par *True*, *False*, le nom d'un champ *Boolean* ou par le nom d'une méthode sans paramètre qui renvoie une valeur *Boolean*. Par exemple :

```
property Nom: TComponentName read FNom write EcritNom stored False;
```

Si la propriété n'a pas de directive **stored**, elle est traitée comme si `stored True` était spécifié.

La directive **default** doit être suivie par une constante du même type que la propriété. Par exemple :

```
property Balise: Longint read FBalise write FBalise default 0;
```

Pour surcharger une valeur **default** héritée sans en spécifier de nouvelle valeur, utilisez la directive **nodefault**. Les directives **default** et **nodefault** ne sont gérées que pour les types scalaires et pour les types ensemble dont les bornes inférieure et supérieure du type de base de l'ensemble sont des valeurs scalaires comprises entre 0 et 31 ; si une telle propriété est déclarée sans **default** ni **nodefault**, elle est traitée comme si **nodefault** était spécifié. Pour les réels, les pointeurs et les chaînes, il y a une valeur par défaut implicite de 0, **nil et** `''` (la chaîne vide), respectivement

Quand on enregistre l'état d'un composant, les spécificateurs de stockage des propriétés publiées du composant sont vérifiés. Si la valeurs en cours de la propriété est différente de sa valeur par défaut (ou s'il n'y a pas de valeur par défaut) et si le spécificateur **stored** a la valeur *True,* alors la valeur de la propriété est enregistrée. Sinon, la valeur de la propriété n'est pas enregistrée.

Remarque Les spécificateurs de stockage ne sont pas gérés pour les propriétés tableau. La directive **default** a une signification différente quand elle est utilisée dans une déclaration de propriété tableau. Voir “Propriétés tableau” à la page 7-21.

Surcharge et redéfinition de propriétés

Une déclaration de propriété qui ne spécifie pas de type est appelée une *redéfinition de propriété*. La redéfinition de propriété permet de modifier la visibilité ou les spécificateurs hérités d’une propriété. La redéfinition la plus simple est constituée uniquement du mot réservé **property** suivi de l’identificateur de la propriété héritée ; cette forme est utilisée pour changer la visibilité d’une propriété. Si, par exemple, une classe ancêtre déclare une propriété comme protégée, une classe dérivée peut la redéclarer dans la section publique ou publiée de la classe. La redéfinition de propriété peut comporter les directives **read**, **write**, **stored**, **default** et **nodefault** ; ces directives redéfinissent la directive correspondante héritée. Une directive redéfinie peut remplacer un spécificateur d’accès hérité, ajouter un spécificateur manquant ou augmenter la visibilité d’une propriété. Elle ne peut pas supprimer un spécificateur d’accès ou réduire la visibilité d’une propriété. Une redéfinition peut comporter la directive **implements** qui s’ajoute à la liste des interfaces implémentées sans en supprimer d’interface héritées.

Les déclarations suivantes illustrent la redéfinition de propriété :

```
type
  TAncetre = class
   ⋮
  protected
    property Taille: Integer read FTaille;
    property Texte: string read LitTexte write EcritTexte;
    property Couleur: TColor read FCouleur write EcritCouleur stored False;
    ⋮
  end;
type
  TDerivee = class(TAncetre)
   ⋮
  protected
    property taille write EcritTaille;
  published
    property Texte;
    property Couleur stored True default clBlue;
    ⋮
  end;
```

La redéfinition de *Taille* ajoute un spécificateur **write** afin de permettre la modification de la propriété. La redéfinition des propriétés *Texte* et *Couleur* fait passer la visibilité des propriétés de protégée à publiée. La redéfinition de la propriété *Couleur* spécifie également que sa valeur doit être enregistrée quand elle n’est pas *clBlue*.

Une redéclaration de propriété qui inclut un identificateur de type masque la propriété héritée au lieu de la redéfinir. Cela signifie qu’une nouvelle propriété

est créée portant le même nom que celle héritée. Toute déclaration de propriété spécifiant un type doit être une déclaration complète et doit donc comporter au moins un spécificateur d'accès.

Si une propriété est masquée ou redéfinie dans une classe dérivée, le substitut de la propriété est toujours statique. C'est-à-dire que c'est le type déclaré (à la compilation) de la variable utilisée pour identifier un objet qui détermine l'interprétation de ses identificateurs de propriété. Ainsi, une fois le code suivant exécuté, la lecture ou l'affectation d'une valeur à *MonObjet.Valeur* appelle *Methode1* ou *Methode2* même si *MonObjet* contient une instance de *TDescendant*. Mais vous pouvez transtyper *MonObjet en TDescendant* pour accéder aux propriétés de la classe dérivée et à leurs spécificateurs d'accès.

```
type
  TAncetre = class
     ⋮
    property Valeur: Integer read Methode1 write Methode2;
  end;

  TDescendant = class(TAncetre)
     ⋮
    property Valeur: Integer read Methode3 write Methode4;
  end;

var MonObjet: TAncetre;
 ⋮
MonObjet := TDescendant.Create;
```

Références de classe

Parfois des opérations sont effectuées sur la classe même et pas sur une instance de la classe (c'est-à-dire sur des objets). Cela se produit, par exemple, quand vous appelez une méthode constructeur utilisant une référence de classe. Vous pouvez toujours désigner une classe spécifique par son nom, mais dans certains cas, il est nécessaire de déclarer des variables ou des paramètres qui prennent des classes pour valeur. Il faut alors utiliser des *types référence de classe*.

Types de référence de classe

Un type référence de classe, appelé parfois une *métaclasse*, est désigné par une construction de la forme :

`class of` *type*

où *type* est un type classe. L'identificateur *type* même désigne une valeur dont le type est `class of` *type*. Si $type_1$ est un ancêtre de $type_2$, alors `class of` $type_2$ est compatible pour l'affectation avec `class of` $type_1$. Donc :

```
type TClass = class of TObject;
var ToutObjet : TClass;
```

déclare une variable appelée *ToutObjet* qui peut contenir une référence sur toute classe. La définition d'un type référence de classe ne peut se faire directement dans une déclaration de variable ou une liste de paramètres. Vous pouvez affecter la valeur **nil** à une variable de tout type référence de classe.

Un exemple d'utilisation de type référence de classe est fourni par la déclaration du constructeur de *TCollection* (dans l'unité *Classes*) :

```
type TCollectionItemClass = class of TCollectionItem;
  ⋮
constructor Create(ItemClass: TCollectionItemClass);
```

Cette déclaration indique que pour créer un objet instance de *TCollection*, il faut transmettre au constructeur le nom d'une classe descendant de *TCollectionItem*.

Les types référence de classe sont utiles quand vous voulez désigner une méthode de classe ou un constructeur virtuel pour une classe ou un objet dont le type réel est inconnu à la compilation.

Constructeurs et références de classe

Il est possible d'appeler un constructeur en utilisant une variable de type référence de classe. Cela permet la construction d'objets dont le type est inconnu à la compilation. Par exemple :

```
type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
    const ControlName: string; X, Y, W, H: Integer): TControl;
begin
  Result := ControlClass.Create(MainForm);
  with Result do
  begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
  end;
end;
```

La fonction *CreateControl* nécessite un paramètre référence de classe pour lui indiquer le type de contrôle à créer. Elle utilise ce paramètre pour appeler le constructeur de la classe. Comme les identificateurs de type classe désignent des valeurs référence de classe, un appel à *CreateControl* peut spécifier l'identificateur de la classe pour en créer une instance. Par exemple :

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Les constructeurs appelés en utilisant des références de classe sont généralement virtuels. L'implémentation du constructeur qui est activée par l'appel dépend du type à l'exécution de la référence de classe.

Opérateurs de classe

Chaque classe hérite de *TObject* les méthodes *ClassType* et *ClassParent* qui renvoient, respectivement, une référence à la classe d'un objet et à l'ancêtre immédiat de l'objet. Ces deux méthodes renvoient une valeur de type *TClass* (où `TClass = class of TObject`), qui peut être transtypée dans un type plus spécifique. Chaque classe hérite également de la méthode *InheritsFrom* qui teste si l'objet pour lequel elle est appelée hérite d'une classe spécifiée. Ces méthodes sont utilisées par les opérateurs **is** et **as**, il est donc rarement nécessaire de les appeler directement.

Opérateur is

L'opérateur **is**, qui effectue une vérification de type dynamique, est utilisé pour vérifier quelle est effectivement la classe d'un objet à l'exécution. L'expression :

objet `is` *classe*

renvoie *True* si *objet* est une instance de la classe désignée par *classe* ou de l'un de ces descendants, et *False* sinon. Si *objet* a la valeur **nil**, le résultat est *False*. Si le type déclaré pour *objet* n'est pas relié à *classe* (c'est-à-dire si les types sont distincts ou si l'un n'est pas l'ancêtre de l'autre), il y a erreur de compilation. Par exemple :

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

Cette instruction transtype une variable en *TEdit* après avoir vérifié que l'objet qu'elle référence est bien une instance de *TEdit* ou de l'un de ses descendants.

Opérateur as

L'opérateur **as** effectue des transtypages avec vérification. L'expression :

objet `as` *classe*

renvoie une référence au même objet que *objet* mais avec le type spécifié par *classe*. A l'exécution, *objet* doit être une instance de la classe désignée par *classe* ou de l'un de ses descendants ou avoir la valeur **nil** ; sinon une exception est déclenchée. Si le type déclaré *objet* n'a pas de lien avec *classe* (c'est-à-dire si les types sont distincts ou si l'un n'est pas l'ancêtre de l'autre), il y a erreur de compilation. Par exemple :

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

Les règles de priorité des opérateurs obligent souvent à placer entre parenthèses un transtypage **as**. Par exemple :

```
(Sender as TButton).Caption := '&Ok';
```

Méthodes de classe

Une méthode de classe est une méthode (autre qu'un constructeur) qui agit sur des classes et pas sur des objets. La définition d'une méthode de classe doit commencer par le mot réservé **class**. Par exemple :

```
type
  TFigure = class
  public
    class function Gestion(Operation: string): Boolean; virtual;
    class procedure ObtenirInfos(var Info: TFigureInfo); virtual;
    ⋮
  end;
```

La déclaration de définition d'une méthode de classe doit également commencer par **class**. Par exemple :

```
class procedure TFigure.ObtenirInfos(var Info: TFigureInfo);
begin
  ⋮
end;
```

Dans la déclaration de définition d'une méthode de classe, l'identificateur *Self* représente la classe où la méthode est appelée (ce peut être un descendant de la classe dans laquelle elle est définie). Si la méthode est appelée dans la classe *C*, alors *Self* est de type `class of` *C*. Vous ne pouvez donc pas utiliser *Self* pour accéder aux champs, propriétés et méthodes normales (les méthodes d'objet). Par contre, vous pouvez l'utiliser pour appeler les constructeurs ou d'autres méthodes de classe.

Une méthode de classe peut être appelée via une référence de classe ou une référence d'objet. Quand elle est appelée via une référence d'objet, la classe de l'objet devient la valeur de *Self*.

Exceptions

Une *exception* est déclenchée quand une erreur ou un autre événement interrompt le déroulement normal d'un programme. L'exception transfère le contrôle à un *gestionnaire d'exceptions,* ce qui vous permet de séparer la logique normale d'exécution du programme de la gestion des erreurs. Comme les exceptions sont des objets, elles peuvent être regroupées en hiérarchies en utilisant l'héritage et de nouvelles exceptions peuvent être ajoutées sans affecter le code existant. Une exception peut véhiculer des informations, par exemple un message d'erreur, depuis le point où elle est déclenchée jusqu'au point où elle est gérée.

Quand une application utilise l'unité *SysUtils*, toutes les erreurs d'exécution sont automatiquement converties en exceptions. Les erreurs qui autrement provoqueraient l'arrêt d'une application (mémoire insuffisante, division par zéro, erreurs de protection générales) peuvent ainsi être interceptées et gérées.

Quand utiliser des exceptions

Les exceptions offrent un moyen élégant d'intercepter les erreurs d'exécution sans arrêter le programme et sans utiliser d'encombrantes instructions conditionnelles. Cependant, la complexité du mécanisme de gestion des exceptions du Pascal Objet, le rend peu efficace et il doit être utilisé judicieusement. Il est possible de déclencher des exceptions pour presque toutes les raisons et de protéger pratiquement n'importe quel bloc de code en l'intégrant dans une instruction **try...except** ou **try...finally**, mais, en pratique, il vaut mieux réserver ces outils à des situations particulières.

La gestion des exceptions convient aux erreurs qui ont peu de chances de se produire, mais dont les conséquences sont quasiment catastrophiques (le crash d'une application, par exemple) ; aux conditions d'erreurs difficiles à tester dans des instructions **if...then** ; et quand vous avez besoin de répondre aux exceptions déclenchées par le système d'exploitation ou par des routines dont le code source n'est pas sous votre contrôle. Les exceptions sont couramment utilisées pour les erreurs matérielles, de mémoire, d'entrée/sortie et du système d'exploitation.

Les instructions conditionnelle sont souvent le meilleur moyen de tester les erreurs. Par exemple, supposons que vous vouliez vous assurer de l'existence d'un fichier avant d'essayer de l'ouvrir. Vous pourriez le faire comme ceci :

```
try
  AssignFile(F, FileName);
  Reset(F);  // déclenche une exception EInOutError si le fichier est introuvable
except
  on Exception do ...
end;
```

Mais, vous pourriez aussi éviter la lourdeur de gestion d'exception en utilisant :

```
if FileExists(FileName) then  // renvoie False si le fichier est introuvable ;
                              // ne déclenche aucune exception
begin
  AssignFile(F, FileName);
  Reset(F);
end;
```

Les *assertions* fournissent un autre moyen de tester une condition booléenne à n'importe quel endroit du code. Quand une instruction *Assert* échoue, le programme s'arrête ou (s'il utilise l'unité *SysUtils*) déclenche une exception *EAssertionFailed*. Les assertions devraient n'être utilisées que pour tester les conditions que vous ne souhaitez pas voir se produire. Pour plus d'informations, voir la procédure *Assert* standard, dans l'aide en ligne.

Déclaration des types exception

Les types exception sont déclarés comme les autres classes. En fait, il est possible d'utiliser comme exception une instance de toute classe. Il est néanmoins préférable de dériver les exceptions de la classe *Exception* définie dans *SysUtils*.

Vous pouvez grouper les exceptions en familles en utilisant l'héritage. Par exemple, les déclarations suivantes de *SysUtils* définissent une famille de types exception pour les erreurs mathématiques :

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

Etant donné ces déclarations, vous pouvez définir un seul gestionnaire d'exceptions *EMathError* qui gère également *EInvalidOp*, *EZeroDivide*, *EOverflow* et *EUnderflow*.

Les classes d'exceptions définissent parfois des champs, des méthodes ou des propriétés qui contiennent des informations supplémentaires sur l'erreur. Par exemple :

```
type EInOutError = class(Exception)
    ErrorCode: Integer;
  end;
```

Déclenchement et gestion des exceptions

Pour créer un objet exception, appelez le constructeur de la classe d'exception à l'intérieur d'une instruction **raise**. Par exemple :

```
raise EMathError.Create;
```

En général, une instruction **raise** a la forme :

raise *objet* at *adresse*

où *objet* et at *adresse* sont tous les deux facultatifs. Si *objet* est omis, l'instruction redéclenche l'exception en cours ; voir "Redéclenchement d'exceptions" à la page 7-33. Quand une *adresse est* spécifiée, c'est généralement un pointeur sur une procédure ou une fonction ; utilisez cette option pour déclencher l'exception depuis un enplacement de la pile antérieur à celui où l'erreur s'est effectivement produite.

Quand une exception est *déclenchée* (c'est-à-dire qu'elle est référencée dans une instruction **raise**), elle est régie par la logique particulière de gestion des exceptions. Une instruction **raise** ne renvoie jamais le contrôle d'une manière normale. Elle transfère à la place le contrôle au gestionnaire d'exceptions le plus proche capable de gérer les exceptions de la classe donnée. Le gestionnaire le plus proche correspond au dernier bloc **try...except** dans lequel le flux d'exécution est entré sans en être encore sorti.

Par exemple, la fonction suivante convertit une chaîne en entier et déclenche une exception *ERangeError* si la valeur résultante est hors de l'intervalle spécifié :

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S);  // StrToInt est déclarée dans SysUtils
```

```
    if (Result < Min) or (Result > Max) then
      raise ERangeError.CreateFmt(
        '%d n''est pas dans l''intervalle spécifié %d..%d',
        [Result, Min, Max]);
  end;
```

Remarquez l'appel de la méthode *CreateFmt* dans l'instruction **raise**. *Exception* et ses descendants ont des constructeurs spéciaux qui proposent d'autres moyens de créer des messages d'exception et des identificateurs de contexte. Pour davantage d'informations, voir l'aide en ligne.

Une exception déclenchée est automatiquement détruite une fois qu'elle a été gérée. N'essayez jamais de détruire manuellement une exception déclenchée.

Remarque Le déclenchement d'une exception dans la section initialisation d'une unité peut ne pas donner le résultat attendu. La gestion normale des exceptions provient de l'unité *SysUtils* qui doit d'abord être initialisée pour que cette gestion soit utilisable. Si une exception se produit lors de l'initialisation, toutes les unités initialisées (dont *SysUtils*) sont finalisées et l'exception est redéclenchée. Alors, l'exception est interceptée et gérée, généralement par l'interruption du programme.

Instructions Try...except

Les exceptions sont gérées dans des instructions **try...except**. Par exemple :

```
try
  X := Y/Z;
except
  on EZeroDivide do GereDivisionParZero;
end;
```

Cette instruction tente de diviser *Y* par *Z* mais appelle la routine appelée *GereDivisionParZero* si une exception *EZeroDivide* est déclenchée.

L'instruction **try...except** a la syntaxe suivante :

`try` *instructions* `except` *blocException* `end`

où *instructions* est une suite d'instructions, délimitée par des points-virgule et *blocException est :*

- une autre suite d'instruction ou
- une suite de gestionnaires d'exceptions, éventuellement suivie par :

  `else` *instructions*

Un gestionnaire d'exception a la forme :

`on` *identificateur*: *type* `do` *instruction*

où *identificateur*: est facultatif (si *identificateur* est précisé, ce doit être un identificateur valide), *type* est le type utilisé pour représenter les exceptions et *instruction* est une instruction quelconque.

Une instruction **try...except** exécute les instructions dans la liste initiale *instructions*. Si aucune exception n'est déclenchée, le bloc exception (*blocException*)

n'est pas pris en compte et le contrôle passe à l'instruction suivante du programme.

Si une exception est déclenchée lors de l'exécution de la liste *instructions* initiale, que ce soit par une instruction **raise** dans la liste *instructions* ou par une procédure ou une fonction appelée dans la liste *instructions*, il va y avoir une tentative de "gestion" de l'exception :

- Si un des gestionnaires du bloc exception ne correspond à l'exception, le contrôle passe au premier d'entre eux. Un gestionnaire d'exceptions "correspond" à une exception si le *type* du gestionnaire est la classe de l'exception ou un ancêtre de cette classe.
- Si aucun gestionnaire correspondant n'est trouvé, le contrôle passe à l'*instruction* de la clause **else** si elle est définie.
- Si le bloc d'exception est simplement une suite d'instructions sans gestionnaire d'exception, le contrôle passe à la première instruction de la liste.

Si aucune de ces conditions n'est respectée, la recherche continue dans le bloc exception de l'avant-dernière instruction **try...except** dans laquelle le flux du programme est entré et n'est pas encore sorti. Si, là encore, il n'y ni gestionnaire approprié, ni clause **else,** ni liste d'instructions, la recherche se propage à l'instruction **en cours try...except** précédente, etc. Si l'instruction **try...except** la plus éloignée est atteinte sans que l'exception soit gérée, le programme s'interrompt.

Quand l'exception est gérée, le pointeur de la pile est ramené en arrière jusqu'à la procédure ou la fonction contenant l'instruction **try...except** où la gestion a lieu et le contrôle d'exécution passe au gestionnaire d'exception exécuté, à la clause **else** ou à la liste d'instructions. Ce processus efface tous les appels de procédure ou de fonction effectués à partir de l'entrée dans l'instruction **try...except** où l'exception est gérée. L'objet exception est alors automatiquement détruit par un appel de son destructeur *Destroy* et le contrôle revient à l'instruction suivant l'instruction **try...except**. Si un appel des procédures standard *Exit*, *Break* ou *Continue* force la sortie du gestionnaire d'exception, l'objet exception est quand même détruit automatiquement.

Dans l'exemple suivant, le premier gestionnaire d'exceptions gère les exceptions division-par-zéro, le second gère les exceptions de débordement et le dernier gère toutes les autres exceptions mathématiques. *EMathError* apparaît en dernier dans le bloc exception car c'est l'ancêtre des deux autres classes d'exception : s'il apparaît en premier, les deux autres gestionnaires ne sont jamais utilisés.

```
try
 ⋮
except
  on EZeroDivide do GereDivisionParZero;
  on EOverflow do GereDebordement;
  on EMathError do GereErreurMath;
end;
```

Un gestionnaire d'exceptions peut spécifier un identificateur avant le nom de la classe exception. Cela déclare l'identificateur représentant l'objet exception

pendant l'exécution de l'instruction suivant **on...do**. La portée de l'identificateur est limitée à celle de l'instruction. Par exemple :

```
try
  ⋮
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

Si le bloc exception spécifie une clause **else**, la clause **else** gère toutes les exceptions qui ne sont pas gérées par les gestionnaires du bloc. Par exemple :

```
try
  ⋮
except
  on EZeroDivide do GereDivisionParZero;
  on EOverflow do GereDebordement;
  on EMathError do GereErreurMath;
else
  GereLesAutres;
end;
```

Ici la clause **else** gère toutes les exceptions qui ne sont pas des erreurs mathématiques (*EMathError)*.

Si le bloc exception ne contient pas de gestionnaires d'exceptions mais une liste d'instructions, cette liste gère toutes les exceptions. Par exemple :

```
try
  ⋮
except
  GereException;
end;
```

Ici la routine *GereException* gère toutes les exceptions se produisant lors de l'exécution des instructions comprises entre **try** et **except**.

Redéclenchement d'exceptions

Quand le mot réservé **raise** apparaît dans un bloc exception sans être suivi d'une référence d'objet, il déclenche l'exception qui était gérée par le bloc. Cela permet à un gestionnaire d'exception de répondre à une erreur d'une manière partielle, puis de *redéclencher* l'exception. Cela est pratique quand une procédure ou une fonction doit "faire le ménage" après le déclenchement d'une exception sans pouvoir gérer complètement l'exception.

Par exemple, la fonction *GetFileList* alloue un objet *TStringList* et le remplit avec les noms de fichiers correspondant au chemin de recherche spécifié :

```
function GetFileList(const Path: string): TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
```

```
      while I = 0 do
      begin
        Result.Add(SearchRec.Name);
        I := FindNext(SearchRec);
      end;
    except
      Result.Free;
      raise;
    end;
  end;
```

GetFileList crée un objet *TStringList* puis utilise les fonctions *FindFirst* et *FindNext* (définies dans *SysUtils*) pour l'initialiser. Si l'initialisation échoue (car le chemin d'initialisation est incorrect ou parce qu'il n'y a pas assez de mémoire pour remplir la liste de chaînes), *c'est GetFileList* qui doit libérer la nouvelle liste de chaînes car l'appelant ne connaît même pas son existence. C'est pour cela que l'initialisation de la liste de chaînes se fait dans une instruction **try...except**. Si une exception a lieu, le bloc exception de l'instruction libère la liste de chaînes puis redéclenche l'exception.

Exceptions imbriquées

Le code exécuté dans un gestionnaire d'exceptions peut lui aussi déclencher et gérer des exceptions. Tant que ces exceptions sont également gérées dans le gestionnaire d'exceptions, elles n'affectent pas l'exception initiale. Par contre, si une exception déclenchée dans un gestionnaire d'exceptions commence à se propager au-delà du gestionnaire, l'exception d'origine est perdue. Ce phénomène est illustré par la fonction *Tan* suivante.

```
type
  ETrigError = class(EMathError);

function Tan(X: Extended): Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Argument incorrect pour Tan');
  end;
end;
```

Si une exception *EMathError* se produit lors de l'exécution de *Tan*, le gestionnaire d'exceptions déclenche une exception *ETrigError*. Comme *Tan* ne dispose pas de gestionnaire pour *ETrigError*, l'exception se propage au-delà du gestionnaire d'exceptions initial, ce qui provoque la destruction de l'objet exception *EMathError*. Ainsi, pour l'appelant, tout se passe comme si la fonction *Tan* avait déclenché une exception *ETrigError*.

Instructions try...finally

Dans certains cas, il est indispensable que certaines parties d'une opération s'effectuent, que l'opération soit ou non interrompue par une exception. Si, par exemple, une routine prend le contrôle d'une ressource, il est souvent important que cette ressource soit libérée quelle que soit la manière dont la routine

s'achève. Vous pouvez, dans ce genre de situations, utiliser une instruction **try...finally**.

L'exemple suivant illustre comment du code qui ouvre et traite un fichier peut garantir que le fichier est fermé, même s'il y a une erreur à l'exécution.

```
Reset(F);
try
 ⋮ // traiter le fichier F
finally
  CloseFile(F);
end;
```

Une instruction **try...finally** a la syntaxe suivante :

`try` *listeInstruction*$_1$ `finally` *listeInstruction*$_2$ `end`

où chaque *listeInstruction* est une suite d'instructions délimitées par des points-virgule. L'instruction **try...finally** exécute les instructions de *listeInstruction*$_1$ (la clause **try**). Si *listeInstruction*$_1$ se termine sans déclencher d'exception, *listeInstruction*$_2$ (la clause **finally**) est exécutée. Si une exception est déclenchée lors de l'exécution de *listeInstruction*$_1$, le contrôle est transféré à *listeInstruction*$_2$; quand *listeInstruction*$_2$ a fini de s'exécuter, l'exception est redéclenchée. Si un appel des procédures *Exit*, *Break* ou *Continue* force la sortie de *listeInstruction*$_1$, *listeInstruction*$_2$ est exécutée automatiquement. Ainsi, la clause **finally** est toujours exécutée quelle que soit la manière dont se termine l'exécution de la clause **try**.

Si une exception est déclenchée sans être gérée par la clause **finally**, cette exception se propage hors de l'instruction **try...finally** et toute exception déjà déclenchée dans la clause **try** est perdue. La clause **finally** doit donc gérer toutes les exceptions déclenchées localement afin de ne pas perturber la propagation des autres exceptions.

Classes et routines standard des exceptions

L'unité *SysUtils* déclare plusieurs routines standard de gestion d'exceptions, dont *ExceptObject*, *ExceptAddr* et *ShowException*. *SysUtils* et d'autres unités contiennent également de nombreuses classes d'exceptions qui dérivent toutes (sauf *OutlineError*) de *Exception*.

La classe *Exception* contient les propriétés *Message* et *HelpContext* qui peuvent être utilisées pour transmettre une description de l'erreur et un identificateur de contexte pour une aide contextuelle. Elle définit également divers constructeurs qui permettent de spécifier la description et l'identificateur de contexte de différentes manières. Pour davantage d'informations, voir l'aide en ligne.

Chapitre

8

Routines standard et Entrées/Sorties

Ce chapitre décrit les entrées/sorties (E/S) de fichier et de texte et décrit les routines de la bibliothèque standard. La plupart des procédures et fonctions décrites ici sont définies dans l'unité *System* qui est implicitement compilée avec chaque application. Les autres sont prédéfinies au niveau du compilateur mais sont traitées comme si elles appartenaient à l'unité *System*.

Certaines routines standard sont définies dans des unités comme *SysUtils* qui doivent être énumérées dans une clause **uses** pour les rendre utilisables dans les programmes. Par contre, vous ne devez pas spécifier *System* dans une clause **uses** ni modifier l'unité *System* ou tenter de la recompiler explicitement.

Pour davantage d'informations sur les routines décrites ici, voir l'aide en ligne.

Entrées et sorties de fichier

Le tableau suivant énumère les routines d'entrées et de sortie.

Tableau 8.1 Procédures et fonctions d'Entrées/Sorties

Procédure ou fonction	Description
Append	Ouvre un fichier texte existant en ajout.
AssignFile	Affecte le nom d'un fichier externe à une variable fichier.
BlockRead	Lit un ou plusieurs enregistrements d'un fichier sans type.
BlockWrite	Ecrit un ou plusieurs enregistrements dans un fichier sans type.
ChDir	Change le répertoire en cours.
CloseFile	Ferme un fichier ouvert.
Eof	Renvoie l'état de fin de fichier d'un fichier.
Eoln	Renvoie l'état de fin de ligne d'un fichier texte.
Erase	Efface un fichier externe.

Tableau 8.1 Procédures et fonctions d'Entrées/Sorties (suite)

Procédure ou fonction	Description
FilePos	Renvoie la position en cours dans un fichier typé ou sans type.
FileSize	Renvoie la taille en cours d'un fichier, ne s'utilise pas pour les fichiers texte.
Flush	Vide le tampon d'un fichier texte en sortie.
GetDir	Renvoie le répertoire en cours dans le lecteur spécifié.
IOResult	Renvoie une valeur entière indiquant l'état de la dernière fonction d'E/S effectuée.
MkDir	Crée un sous-répertoire.
Read	Lit une ou plusieurs valeurs d'un fichier dans une ou plusieurs variables.
Readln	Fait la même chose que *Read* puis passe au début de la ligne suivante du fichier texte.
Rename	Renomme un fichier externe.
Reset	Ouvre un fichier existant.
Rewrite	Crée et ouvre un nouveau fichier.
RmDir	Supprime un sous-répertoire vide.
Seek	Place la position en cours d'un fichier typé ou non sur le composant spécifié. Ne s'utilise pas avec les fichiers texte.
SeekEof	Renvoie l'état de fin de fichier d'un fichier texte.
SeekEoln	Renvoie l'état de fin de ligne d'un fichier texte.
SetTextBuf	Affecte un tampon d'E/S à un fichier texte.
Truncate	Tronque un fichier avec ou sans type à la position en cours dans le fichier.
Write	Ecrit une ou plusieurs valeurs dans un fichier.
Writeln	Fait la même chose que *Write* puis écrit un marqueur de fin de ligne dans le fichier texte.

Une variable fichier est une variable dont le type est un type fichier. Il y a trois catégories de type fichier : *typé*, *texte* et *non typé*. La syntaxe est la même pour déclarer des types fichier, celle indiquée dans “Types fichier” à la page 5-26.

Avant de pouvoir utiliser une variable fichier, il faut l'associer à un fichier externe en appelant la procédure *AssignFile*. Un fichier externe est généralement un fichier disque portant un nom mais ce peut également être un périphérique comme le clavier ou l'écran. Le fichier externe stocke des informations écrites dans le fichier ou fournit les informations lues dans le fichier.

Une fois établie l'association avec un fichier externe, la variable fichier doit être “ouverte” pour la préparer à des entrées ou des sorties. Un fichier existant peut être ouvert via la procédure *Reset*, un nouveau fichier peut être créé et ouvert via la procédure *Rewrite*. Les fichiers texte ouverts avec *Reset* sont en lecture seule et les fichiers texte ouverts avec *Rewrite* et *Append* sont en écriture seulement. Les fichiers typés et les fichiers non typés permettent toujours la lecture et l'écriture qu'ils soient ouverts avec *Reset* ou *Rewrite*.

Tout fichier est une suite linéaire de composants, chacun ayant le type de composant (ou d'enregistrement) du fichier. Ces composants sont numérotés à partir de zéro.

L'accès aux fichiers se fait normalement de manière séquentielle. C'est-à-dire que si un composant est lu en utilisant la procédure standard *Read* ou écrit en utilisant la procédure standard *Write*, la position en cours dans le fichier passe au composant suivant dans l'ordre numérique. Il est possible d'accéder directement aux fichiers typés et non typés en utilisant la procédure standard *Seek*, qui déplace la position en cours dans le fichier sur le composant spécifié. Les fonctions standard *FilePos* et *FileSize* peuvent être utilisées pour déterminer la position en cours dans le fichier et la taille du fichier en cours.

Quand un programme termine le traitement d'un fichier, le fichier doit être fermé en utilisant la procédure standard *CloseFile*. Une fois un fichier fermé, son fichier externe est actualisé. La variable fichier peut alors être associée à un autre fichier externe.

Par défaut, tous les appels des procédures et fonctions standard d'entrées/sorties sont automatiquement vérifiés pour détecter les erreurs et, si une erreur s'est produite, une exception est déclenchée (ou le programme s'arrête si la gestion des exceptions n'est pas activée). Cette vérification automatique peut être activée ou désactivée en utilisant les directives de compilation **{$I+}** et **{$I–}**. Quand la vérification des E/S est désactivée (c'est-à-dire quand une procédure ou fonction est compilée à l'état **{$I–}**), une erreur d'E/S ne provoque pas le déclenchement d'une exception ; pour vérifier le résultat d'une opération d'E/S, vous devez alors appeler la fonction standard *IOResult*.

Vous devez appeler la fonction *IOResult* pour effacer une erreur, même si l'erreur ne vous concerne pas. Si vous n'effacez pas l'erreur et si **{$I+}** est l'état en cours, la prochaine fonction d'E/S appelée échoue avec l'erreur *IOResult* en souffrance.

Fichier texte

Cette section décrit brièvement les entrées/sorties utilisant des variables fichier ayant le type standard *Text*.

A l'ouverture d'un fichier texte, le fichier externe est interprété d'une manière particulière : il est supposé représenter une suite de caractères formatés en lignes. Chaque ligne est terminée par un marqueur de fin de ligne (un caractère de retour-chariot éventuellement suivi d'un caractère de passage à la ligne). Le type *Text* est distinct du type `file of Char`.

Les fichiers texte utilisent des variantes de *Read* et *Write* qui permettent de lire et d'écrire des valeurs qui ne sont pas de type *Char*. De telles valeurs sont automatiquement converties vers ou depuis une représentation texte. Par exemple, `Read(F, I)`, où *I* est une variable de type *Integer*, lit une suite de chiffres et l'interprète comme un entier décimal et la stocke dans *I*.

Il y a deux variables d'entrées/sorties texte standard, *Input* et *Output*. La variable fichier standard *Input* est un fichier en lecture seule associé à l'entrée standard du système d'exploitation (c'est généralement le clavier). La variable fichier texte standard *Output* est un fichier en écriture seulement associée à la sortie standard du système d'exploitation (c'est généralement l'écran). Avant le début d'une

application, les variables *Input* et *Output* sont automatiquement ouvertes et les instructions suivantes sont exécutées :

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Remarque Les variables d'E/S orientées texte ne sont disponibles que dans les applications console. Ce sont les applications compilées avec l'option “Générer application console” cochée dans la page Lieur de la boîte de dialogue Options de projet ou en utilisant l'option de ligne de commande **-cc** du compilateur. Dans une application graphique (non console), toute tentative de lire ou d'écrire en utilisant *Input* ou *Output* provoque une erreur d'E/S.

Certaines des routines standard d'E/S fonctionnent avec des fichiers texte qui n'ont pas nécessairement besoin d'avoir une variable fichier spécifiée explicitement comme paramètre. Si le paramètre fichier est omis, *les variables Input* ou *Output* sont utilisées par défaut selon que la routine est une routine d'entrée ou de sortie. Par exemple, `Read(X)` correspond à `Read(Input, X)` et `Write(X)` correspond à `Write(Output, X)`.

Si vous spécifiez un fichier lors de l'appel d'une de ces routines d'entrée ou de sortie fonctionnant sur les fichiers texte, le fichier doit être associé à un fichier externe en utilisant *AssignFile* et ouvert en utilisant *Reset*, *Rewrite* ou *Append*. Une exception est déclenchée si vous transmettez un fichier ouvert avec *Reset* à une routine de sortie. De même, une exception est déclenchée si vous transmettez un fichier ouvert avec *Rewrite* ou *Append* à une routine d'entrée.

Fichiers sans type

Les fichiers non typés sont des canaux d'E/S de bas niveau utilisés essentiellement pour accéder à des fichiers disque sans tenir compte de leur type ou de leur structure. Un fichier non typé est déclaré avec le mot **file** seul. Par exemple :

```
var FichierDonnees: file;
```

Pour les fichiers non typés, les procédures *Reset* et *Rewrite* autorisent un paramètre supplémentaire afin de spécifier la taille de l'enregistrement utilisé pour les transferts de données. Pour des raisons historiques, la taille d'enregistrement par défaut est de 128 octets. Seul l'enregistrement de taille 1 permet de refléter exactement la taille de tous les fichiers (en effet, il ne peut pas y avoir d'enregistrement partiel quand la taille d'enregistrement est de 1).

A l'exception de *Read* et *Write*, toutes les procédures et fonctions standard des fichiers typés fonctionnent également avec les fichiers non typés. Au lieu de *Read* et *Write*, deux procédures, *BlockRead* et *BlockWrite*, sont utilisées pour des transferts de données rapides.

Pilotes de périphérique de fichiers texte

Vous pouvez définir vos propres pilotes de périphérique de fichier texte dans des programmes. Un pilote de périphérique de fichier texte est un ensemble de quatre fonctions qui implémentent complètement une interface entre le système de fichier Pascal Objet et un périphérique.

Les quatre fonctions qui définissent entièrement un pilote de périphérique sont *Open*, *InOut*, *Flush* et *Close*. Les en-têtes de ces fonctions ont la forme :

```
function DeviceFunc(var F: TTextRec): Integer;
```

où *DeviceFunc* est le nom de la fonction (c'est-à-dire *Open*, *InOut*, *Flush* ou *Close*). Pour plus d'informations sur le type *TTextRec* type, voir l'aide en ligne. La valeur renvoyée par une fonction d'interface de périphérique devient la valeur renvoyée par *IOResult*. Si la valeur renvoyée est zéro, l'opération a réussi.

Pour associer les fonctions d'interface de périphérique à un fichier spécifique, vous devez écrire une procédure *Assign* personnalisée. La procédure *Assign* doit affecter l'adresse des quatre fonctions d'interface de périphérique aux quatre pointeurs de fonction d'une variable fichier texte. De plus, elle doit stocker la constante “magique” *fmClosed* dans le champ *Mode*, stocker la taille du tampon du fichier texte dans *BufSize*, stocker un pointeur sur le tampon du fichier texte dans *BufPtr et* effacer la chaîne *Name*.

En supposant que les quatre fonctions d'interface de périphérique s'appellent *DevOpen*, *DevInOut*, *DevFlush* et *DevClose*, la procédure *Assign* peut alors avoir le contenu suivant :

```
procedure AssignDev(var F: Text);
begin
  with TTextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
  end;
end;
```

Les fonctions d'interface de périphérique peuvent utiliser le champ *UserData* de l'enregistrement du fichier pour stocker des informations privées. Ce champ n'est jamais modifié par le système de fichiers du produit.

Fonctions de périphérique

Les fonctions constituant un pilote de périphérique de fichier texte sont décrites ci-dessous.

Fonction Open

La fonction *Open* est appelée par les procédures standard *Reset*, *Rewrite* et *Append* pour ouvrir un fichier texte associé à un périphérique. En entrée, le champ mode *Mode* contient *fmInput*, *fmOutput* ou *fmInOut* pour indiquer si la fonction *Open* a été appelée par *Reset*, *Rewrite* ou *Append*.

La fonction *Open* prépare le fichier en entrée ou en sortie selon la valeur de *Mode*. Si *Mode* spécifie *fmInOut* (ce qui indique que *Open* a été appelée par *Append*), sa valeur doit être changée en *fmOutput* avant la fin de *Open*.

Open est toujours appelée avant toutes les autres fonctions d'interface de périphérique. De ce fait, *AssignDev* n'initialise que le champ *OpenFunc*, laissant l'initialisation des autres vecteurs à *Open*. En se basant sur la valeur de *Mode*, *Open* peut ensuite installer des pointeurs sur les fonctions d'entrées ou de sortie. Cela évite aux fonctions *InOut*, *Flush* et à la procédure *CloseFile* d'avoir à déterminer le mode en cours.

Fonction InOut

La fonction *InOut* est appelée par les routines standard *Read*, *Readln*, *Write*, *Writeln*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln* et *CloseFile* dès qu'une entrée ou une sortie sur le périphérique est nécessaire.

Si *Mode* a la valeur *fmInput*, la fonction *InOut* lit jusqu'à *BufSize* caractères dans `BufPtr^` et renvoie le nombre de caractères lus dans *BufEnd*. De plus, elle stocke zéro dans *BufPos*. Si la fonction *InOut* renvoie zéro dans *BufEnd* comme résultat d'une demande de lecture, *Eof* prend la valeur *True* pour le fichier.

Si *Mode* a la valeur *fmOutput*, la fonction *InOut* écrit *BufPos* caractères de `BufPtr^` et renvoie zéro dans *BufPos*.

Fonction Flush

La fonction *Flush* est appelée à la fin de chaque *Read*, *Readln*, *Write* et *Writeln*. Elle peut éventuellement vider le tampon du fichier texte.

Si *Mode* a la valeur *fmInput*, la fonction *Flush* peut placer zéro dans *BufPos* et dans *BufEnd* afin de vider les caractères restants (non lus) du tampon. Cette caractéristique est rarement utilisée.

Si *Mode* a la valeur *fmOutput*, la fonction *Flush* peut écrire le contenu du tampon exactement comme la fonction *InOut,* ce qui garantit que le texte écrit dans le périphérique apparaît bien immédiatement sur le périphérique. Si *Flush* ne fait rien, le texte n'apparaît pas sur le périphérique tant que le tampon n'est pas plein ou que le fichier n'est pas fermé.

Fonction Close

La fonction *Close* est appelée par la procédure standard *CloseFile* afin de fermer un fichier texte associé à un périphérique. Les procédures *Reset*, *Rewrite* et *Append* appellent également *Close* si le fichier qu'elles ouvrent est déjà ouvert. Si *Mode* a la valeur *fmOutput*, le système de fichier appelle la fonction *InOut avant d'appeler Close*, afin de garantir que tous les caractères ont bien été écrits sur le périphérique.

Gestion des chaînes à zéro terminal

La syntaxe étendue du Pascal Objet permet aux procédures standard *Read*, *Readln*, *Str* et *Val* de s'appliquer à des tableaux de caractères d'indice de base zéro et permet aux procédures standard *Write*, *Writeln*, *Val*, *AssignFile* et *Rename* de s'appliquer à des tableaux de caractères d'indice de base zéro et à des pointeurs de caractère. De plus, les fonctions suivantes permettent de manipuler des chaînes à zéro terminal. Pour plus d'informations sur les chaînes à zéro terminal, voir “Manipulation des chaînes à zéro terminal” à la page 5-14.

Tableau 8.2 Fonctions de manipulation des chaînes à zéro terminal

Fonction	Description
StrAlloc	Alloue sur le tas un tampon de caractères d'une taille donnée.
StrBufSize	Renvoie la taille du tampon de caractères alloué sur le tas en utilisant *StrAlloc* ou *StrNew*.
StrCat	Concatène deux chaînes.
StrComp	Compare deux chaînes.
StrCopy	Copie une chaîne.
StrDispose	Libère un tampon de caractères alloué en utilisant *StrAlloc* ou *StrNew*.
StrECopy	Copie une chaîne et renvoie un pointeur sur la fin de la chaîne.
StrEnd	Renvoie un pointeur sur la fin de la chaîne.
StrFmt	Formate une ou plusieurs valeurs dans une chaîne.
StrIComp	Compare deux chaînes sans tenir compte des différences majuscules/minuscules.
StrLCat	Concatène deux chaînes avec une longueur maximum donnée pour la chaîne résultante.
StrLComp	Compare deux chaînes sur une longueur maximum donnée.
StrLCopy	Copie une chaîne jusqu'à une longueur maximum donnée.
StrLen	Renvoie la longueur d'une chaîne.
StrLFmt	Formate une ou plusieurs valeurs dans une chaîne avec une longueur maximum donnée.
StrLIComp	Compare deux chaînes sur une longueur maximum donnée sans tenir compte des différences majuscules/minuscules.
StrLower	Convertit une chaîne en minuscules.
StrMove	Déplace un bloc de caractères d'une chaîne dans une autre.
StrNew	Alloue une chaîne sur le tas.

Tableau 8.2 Fonctions de manipulation des chaînes à zéro terminal (suite)

Fonction	Description
StrPCopy	Copie une chaîne Pascal dans une chaîne à zéro terminal.
StrPLCopy	Copie une chaîne Pascal dans une chaîne à zéro terminal avec une longueur maximum donnée.
StrPos	Renvoie un pointeur sur la première occurrence d'une sous-chaîne donnée dans une chaîne.
StrRScan	Renvoie un pointeur sur la dernière occurrence d'un caractère donné dans une chaîne.
StrScan	Renvoie un pointeur sur la première occurrence d'un caractère donné dans une chaîne.
StrUpper	Convertit une chaîne en majuscules.

Les fonctions standard de manipulation de chaînes ont également un équivalent multi-octets qui implémente aussi le tri des caractères en tenant compte de la localisation. Le nom des fonctions multi-octets commence par *Ansi-*. Ainsi, la version multi-octets de *StrPos* est *AnsiStrPos*. La gestion des caractères multi-octets dépend du système d'exploitation et se base sur la localisation en cours

Chaînes de caractères étendus

L'unité *System* propose trois fonctions *WideCharToString*, *WideCharLenToString* et *StringToWideChar* que vous pouvez utiliser pour convertir des chaînes de caractères étendus à zéro terminal en chaînes longues de caractères simples ou double-octets.

Pour davantage d'informations sur les chaînes de caractères étendus, voir “A propos des jeux de caractères étendus” à la page 5-14.

Autres routines standard

Le tableau suivant énumère les fonctions et procédures des bibliothèques Borland qui sont fréquemment utilisées. Il ne constitue pas un inventaire exhaustif de toutes les routines standard. Pour davantage d'informations sur ces routines, voir l'aide en ligne.

Tableau 8.3 Autres routines standard

Procédure ou fonction	Description
Abort	Termine le processus sans indiquer d'erreur.
Addr	Renvoie un pointeur sur un objet spécifié.
AllocMem	Alloue un bloc de mémoire et initialise chaque octet à zéro.
ArcTan	Calcule l'arc-tangente d'un nombre donné.
Assert	Teste si une expression booléenne vaut *True*.
Assigned	Teste si un pointeur ou une variable procédure vaut **nil** (non assigné).

Tableau 8.3 Autres routines standard (suite)

Procédure ou fonction	Description
Beep	Génère un bip standard en utilisant le haut-parleur de l'ordinateur.
Break	Force la sortie d'une instruction **for**, **while** ou **repeat**.
ByteToCharIndex	Renvoie la position dans une chaîne d'un caractère contenant un octet spécifié.
Chr	Renvoie le caractère correspondant à une valeur spécifiée.
Close	Termine l'association entre une variable fichier et un fichier externe.
CompareMem	Effectue une comparaison binaire entre deux images mémoire.
CompareStr	Compare des chaînes en tenant compte des différences majuscules/minuscules.
CompareText	Compare des chaînes par valeur scalaire en ne tenant pas compte des différences majuscules/minuscules.
Continue	Renvoie sur l'itération suivante dans des instructions **for**, **while** ou **repeat**.
Copy	Renvoie une sous-chaîne d'une chaîne ou un segment d'un tableau dynamique.
Cos	Calcule le cosinus d'un angle donné.
CurrToStr	Convertit une variable monétaire en chaîne.
Date	Renvoie la date en cours.
DateTimeToStr	Convertit une variable de type *TDateTime* en chaîne.
DateToStr	Convertit une variable de type *TDateTime* en chaîne.
Dec	Décrémente une variable scalaire.
Dispose	Libère la mémoire allouée à une variable dynamique.
ExceptAddr	Renvoie l'adresse à laquelle a été déclenchée l'exception en cours.
Exit	Sort de la procédure en cours.
Exp	Calcule l'exponentielle d'une valeur donnée.
FillChar	Remplit des octets contigus avec la valeur spécifiée.
Finalize	Libère une variable allouée dynamiquement.
FloatToStr	Convertit une valeur flottante en chaîne.
FloatToStrF	Convertit une valeur flottante en chaîne en utilisant le format spécifié.
FmtLoadStr	Renvoie une chaîne de sortie formatée en utilisant une chaîne de format ressource.
FmtStr	Assemble une chaîne formatée à partir d'une série de tableaux.
Format	Assemble une chaîne à partir d'une chaîne de format et d'une série de tableaux.
FormatDateTime	Formate une valeur date-heure.
FormatFloat	Formate une valeur à virgule flottante.
FreeMem	Libère une variable dynamique.
GetMem	Crée une variable dynamique et un pointeur sur l'adresse du bloc.
GetParentForm	Renvoie la fiche ou la page de propriétés qui contient le contrôle spécifié.
Halt	Provoque un arrêt anormal du programme.

Tableau 8.3 Autres routines standard (suite)

Procédure ou fonction	Description
Hi	Renvoie l'octet de poids fort d'une expression comme valeur non signée.
High	Renvoie la plus grande valeur de l'étendue d'un intervalle, d'un tableau ou d'une chaîne.
Inc	Incrémente une variable scalaire.
Initialize	Initialise une variable allouée dynamiquement.
Insert	Insère une sous-chaîne dans une chaîne à la position spécifiée.
Int	Renvoie la partie entière d'un nombre réel.
IntToStr	Convertit un entier en chaîne.
Length	Renvoie la longueur d'une chaîne ou d'un tableau.
Lo	Renvoie l'octet de poids faible d'une expression comme valeur non signée.
Low	Renvoie la plus petite valeur de l'étendue d'un intervalle, d'un tableau ou d'une chaîne.
LowerCase	Convertit une chaîne ASCII en minuscules.
MaxIntValue	Renvoie la plus grande valeur signée dans un tableau d'entiers.
MaxValue	Renvoie la plus grande valeur signée dans un tableau.
MinIntValue	Renvoie la plus petite valeur signée dans un tableau d'entiers.
MinValue	Renvoie la plus petite valeur signée dans un tableau.
New	Crée une nouvelle variable dynamique et la référence avec le pointeur spécifié.
Now	Renvoie l'heure et la date en cours.
Ord	Renvoie le rang d'une expression de type scalaire.
Pos	Renvoie l'indice dans une chaîne du premier caractère d'une sous-chaîne spécifiée.
Pred	Renvoie le prédécesseur d'une valeur scalaire.
Ptr	Convertit l'adresse spécifiée en pointeur.
Random	Génère des valeurs aléatoires dans l'intervalle spécifié.
ReallocMem	Réalloue une variable dynamique.
Round	Renvoie la valeur d'un réel arrondie à l'entier le plus proche.
SetLength	Définit la longueur dynamique d'une variable chaîne ou tableau.
SetString	Définit le contenu et la longueur d'une chaîne donnée.
ShowException	Affiche un message d'exception avec son adresse.
ShowMessage	Affiche une boîte message avec une chaîne non formatée et le bouton OK.
ShowMessageFmt	Affiche une boîte message avec une chaîne formatée et le bouton OK.
Sin	Renvoie le sinus de l'angle spécifié en radians.
SizeOd	Renvoie le nombre d'octets occupés par une variable ou un type.
Sqr	Renvoie le carré d'un nombre.
Sqrt	Renvoie la racine carrée d'un nombre.
Str	Formate une chaîne et la renvoie dans une variable.
StrToCurr	Convertit une chaîne en valeur monétaire.

Tableau 8.3 Autres routines standard (suite)

Procédure ou fonction	Description
StrToDate	Convertit une chaîne en valeur date (*TDateTime*).
StrToDateTime	Convertit une chaîne en valeur *TDateTime*.
StrToFloat	Convertit une chaîne en valeur à virgule flottante.
StrToInt	Convertit une chaîne en valeur entière.
StrToTime	Convertit une chaîne au format heure (*TDateTime*).
StrUpper	Renvoie une chaîne en majuscules.
Succ	Renvoie le successeur d'une valeur scalaire.
Sum	Renvoie la somme des éléments d'un tableau.
Time	Renvoie l'heure en cours.
TimeToStr	Convertit une variable de type *TDateTime* en chaîne.
Trunc	Tronque un nombre réel en un entier.
UniqueString	Garantit qu'une chaîne n'a qu'une seule référence. (La chaîne peut être copiée pour produire une seule référence.)
UpCase	Convertit un caractère en majuscules.
UpperCase	Renvoie une chaîne en majuscules.
VarArrayCreate	Crée un tableau variant.
VarArrayDimCount	Renvoie le nombre de dimensions d'un tableau variant.
VarARrayHighBound	Renvoie la borne supérieure d'une dimension d'un tableau variant.
VarArrayLock	Verrouille un tableau variant et renvoie un pointeur sur les données.
VarArrayLowBound	Renvoie la borne inférieure d'une dimension d'un tableau variant.
VarArrayOf	Crée et remplit un tableau variant à une dimension.
VarArrayRedim	Redimensionne un tableau variant.
VarArrayRef	Renvoie une référence au tableau variant transmis.
VarArrayUnlock	Déverrouille un tableau variant.
VarAsType	Convertit un variant dans le type spécifié.
VarCast	Convertit un variant dans le type spécifié et stocke le résultat dans une variable.
VarClear	Efface un variant.
CarCopy	Copie un variant.
VarToStr	Convertit un variant en chaîne.
VarType	Renvoie le code de type du variant spécifié.

Pour des informations sur les chaînes de format, voir “Chaînes de format” dans l'aide en ligne.

Partie

II

Sujets spéciaux

Les chapitres de la partie II traitent de caractéristiques spécialisées du langage et abordent des sujets plus techniques. Ces chapitres sont :

- Chapitre 9, “Bibliothèques et paquets”
- Chapitre 10, “Interfaces d’objets”
- Chapitre 11, “Gestion de la mémoire”
- Chapitre 12, “Contrôle des programmes”
- Chapitre 13, “Code assembleur en ligne”

Chapitre

9

Bibliothèques et paquets

Une bibliothèque à chargement dynamique est une bibliothèque de liaison dynamique (DLL) sous Windows ou une bibliothèque d'objet partagé sous Linux. C'est une collection de routines, pouvant être appelées par des applications ou par d'autres DLL ou objets partagés. Comme les unités, les bibliothèques à chargement dynamique contiennent du code partagé ou des ressources. Mais, ce type de bibliothèque est un exécutable compilé séparément qui est lié, à l'exécution, aux programmes qui les utilisent.

Pour les distinguer des exécutables autonomes, les fichiers sous Windows contenant des DLL compilées ont l'extension .DLL. Sous Linux les fichiers contenant des objets partagés ont l'extension .so. Les programmes Pascal Objet peuvent appeler des DLL ou des objets partagés écrits dans d'autres langages et les applications écrites dans d'autres langages peuvent appeler des DLL et des objets partagés écrits en Pascal Objet.

Appel de bibliothèques à chargement dynamique

Vous pouvez appeler les routines du système d'exploitation directement, mais elles ne sont liées à votre application qu'à l'exécution. Cela signifie que la bibliothèque n'a pas besoin d'être présente lors de la compilation du programme. Cela signifie également qu'il n'y a pas vérification à la compilation de la validité d'une importation de routine.

Avant de pouvoir appeler les routines définies dans un objet partagé, il faut les *importer*. Cela peut se faire de deux manières : en déclarant une procédure ou une fonction **external** ou en faisant un appel direct au système d'exploitation. Quelle que soit la méthode utilisée, les routines ne sont pas liées à l'application avant l'exécution.

Pascal Objet ne gère pas l'importation de variables depuis les bibliothèques.

Chargement statique

La manière la plus simple d'importer une procédure ou une fonction est de la déclarer en utilisant la directive **external**. Par exemple :

```
Sous Windows: procedure FaireQuelquechose; external 'MALIB.DLL';

Sous Linux:   procedure FaireQuelquechose; external 'malib.so';
```

Si vous incluez cette déclaration dans un programme, MALIB.DLL (sous Windows) ou malib.so (sous Linux) est chargée au moment du démarrage du programme. Durant toute l'exécution du programme, l'identificateur *FaireQuelquechose* désigne toujours le même point d'entrée dans la même bibliothèque partagée.

La déclaration des routines importées peut se placer directement dans le programme ou l'unité qui les appelle. Pour simplifier la maintenance, vous pouvez aussi rassembler des déclarations **external** dans une “unité d'importation” séparée qui contient également les constantes et les types nécessaires pour communiquer avec la bibliothèque. D'autres modules utilisant l'unité d'importation peuvent appeler toutes les routines qui y sont déclarées.

Pour davantage d'informations sur les déclarations **external**, voir “Déclarations externes” à la page 6-7.

Chargement dynamique

Vous pouvez accéder à des routines d'une bibliothèque via des appels directs de fonctions des bibliothèques système, dont *LoadLibrary*, *FreeLibrary* et *GetProcAddress.* Sous Windows, ces fonctions sont déclarées dans Windows.pas ; sous Linux, elles sont implémentées pour des raisons de compatibilité dans SysUtils.pas ; les routines système de Linux sont *dlopen*, *dlclose* et *dlsym* (toutes déclarées dans l'unité *Libc* de Kylix ; voir les pages man pour plus d'informations). Dans ce cas, utilisez des variables de type procédure pour désigner les routines importées.

Par exemple, sous Windows ou Linux :

```
uses Windows, ...; {sous Linux, remplacer Windows par SysUtils }

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ⋮
begin
  Handle := LoadLibrary('libraryname');
```

```
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('Il est ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
  end;
end;
```

Quand vous importez ainsi des routines, la bibliothèque n'est pas chargée avant l'exécution du code contenant l'appel de *LoadLibrary*. La bibliothèque est déchargée ultérieurement par un appel de *FreeLibrary*. Cela vous permet d'économiser la mémoire et d'exécuter le programme, même si certaines bibliothèques utilisées sont absentes.

Ce même exemple peut aussi être écrit sous Linux :

```
uses Libc, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Pointer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ⋮
begin
  Handle := dlopen('datetime.so', RTLD_LAZY);
  if Handle <> 0 then
  begin
    @GetTime := dlsym(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('Il est ', Hour, ':', Minute, ':', Second);
    end;
    dlclose(Handle);
  end;
end;
```

Dans ce cas, lors de l'importation des routines, l'objet partagé n'est pas chargé avant que le code contenant l'appel de *dlopen* s'exécute. L'objet partagé est déchargé plus tard par l'appel de *dlclose*. Cela vous permet également de

conserver la mémoire et d'exécuter votre programme même si certains des objets partagés qu'il utilise ne sont pas présents.

Ecriture de bibliothèques à chargement dynamique

La source principale d'une bbb est identique à celle d'un programme, mais elle commence par le mot réservé **library** (au lieu de **program**).

Seule les routines qu'une bibliothèque exporte de façon explicite peuvent être importées par d'autres bibliothèques ou programmes. L'exemple suivant propose une bibliothèque qui exporte deux fonctions, *Min* et *Max*.

```
library MinMax;

function Min(X, Y: Integer): Integer; stdcall;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; stdcall;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min,
  Max;

begin
end.
```

Si vous voulez utiliser votre bibliothèque dans des applications écrites dans d'autres langages, il est plus prudent de spécifier la directive **stdcall** dans la déclaration des fonctions exportées. Certains langages ne gèrent pas la convention d'appel par défaut **register du** Pascal Objet.

Il est possible de concevoir une bibliothèque à partir de plusieurs unités. Dans ce cas, le fichier source de la bibliothèque est souvent réduit à une clause **uses**, une clause **exports** et le code d'initialisation. Par exemple :

```
library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors,
  DoneEditors name Done,
  InsertText name Insert,
  DeleteSelection name Delete,
  FormatSelection,
  PrintSelection name Print,
   ⋮
  SetErrorHandler;

begin
  InitLibrary;
end.
```

Vous pouvez placer des clauses **exports** dans la section **interface** ou **implementation** d'une unité. Toute bibliothèque dont la clause **uses** inclut une telle unité exporte automatiquement les routines listées dans la clause **exports** de l'unité — sans qu'elle ait besoin d'avoir sa propre clause **exports**.

La directive **local**, qui marque les routines comme indisponibles pour l'exportation, est spécifique à la plate-forme et n'a pas d'effet en programmation Windows.

Sous Linux, la directive **local** fournit une légère optimisation de performance pour les routines qui sont compilées dans une bibliothèque mais pas exportées. Cette directive peut être spécifiée pour des procédures et fonctions autonomes, mais pas pour des méthodes. Une routine déclarée avec **local** — par exemple,

```
function Contraband(I: Integer): Integer; local;
```

— ne réactualise pas le registre EBX et ici

- ne peut pas être exportée d'une bibliothèque ;
- ne peut pas être déclarée dans la section **interface** de l'unit ;
- ne doit pas avoir son adresse isuue d'une variable de type procédural ni lui être affectée ;
- si c'est une routine en assembleur pur, ne peut pas être appelée depuis une autre unité sauf si l'appelant définit EBX.

Clause exports

Une routine est exportée si elle est énumérée dans une clause **exports** qui a la forme suivante :

exports *entrée*$_1$, ..., *entrée*$_n$;

où chaque *entrée* est composée du nom d'une procédure ou d'une fonction (qui doit être déclarée avant la clause **exports**) suivi par une liste de paramètres (seulement si la routine est surchargée) et un spécificateur facultatif **name**. Vous pouvez qualifier le nom de la procédure ou de la fonction avec le nom d'une unité.

Les entrées peuvent également contenir la directive **resident** qui, conservée dans un souci de compatibilité ascendante, n'est pas prise en compte par le compilateur.

Sous Windows seulement, un spécificateur **index** est constitué de la directive **index** suivie par une constante numérique comprise entre 1 et 2 147 483 647. (Pour que les programmes soient plus efficaces, utilisez des valeurs basses.) Si une entrée n'a pas de spécificateur **index**, un numéro est automatiquement affecté à la routine dans la table d'exportation.

Remarque L'utilisation de spécificateurs **index**, qui sont supportés uniquement pour des raisons de compatibilité ascendante, est déconseillée et peut provoquer des problèmes pour d'autres outils de développement.

Un spécificateur **name** est constitué de la directive **name** suivie par une constante chaîne. Si une entrée n'a pas de spécificateur **name**, la routine est

exportée sous le nom utilisé pour la déclarer, avec la même orthographe et la même casse. Utilisez une clause **name** quand vous voulez exporter une routine sous un nom différent. Par exemple :

```
exports
  FaireQuelquechoseABC index 1 name 'FaireQuelquechose';
```

Quand vous exportez une fonction ou une procédure surchargée depuis un objet partagé, vous devez spécifier sa liste de paramètres dans la clause **exports**. Par exemple,

```
exports
  Divide(X, Y: Integer) name 'Divide_Ints',
  Divide(X, Y: Real) name 'Divide_Reals';
```

Sous Windows, n'insérez pas de spécificateurs **index** dans les entrées des routines surchargées.

Une clause **exports** peut apparaître n'importe où et à plusieurs reprises dans la partie déclaration d'un programme ou d'une bibliothèque, ou dans la section **interface** ou **implementation** d'une unité. Les programmes contiennent rarement la clause **exports**.

Code d'initialisation d'une bibliothèque

Les instructions placées dans le bloc d'une bibliothèque constituent le *code d'initialisation de la bibliothèque*. Ces instructions sont exécutées une seule fois lors du chargement de la bibliothèque. Elles effectuent généralement l'initialisation de variables ou le recensement de classes Windows. Le code d'initialisation d'une bibliothèque peut également installer une procédure de sortie en utilisant la variable *ExitProc*, comme indiquée dans "Procédures de sortie" à la page 12-5.

Le code d'initialisation d'une bibliothèque peut signaler une erreur en affectant à la variable *ExitCode* une valeur non nulle. *ExitCode*, déclarée dans l'unité *System*, a la valeur par défaut zéro qui indique la réussite de l'initialisation. Si le code d'initialisation d'une bibliothèque affecte à *ExitCode* une valeur différente, la bibliothèque est déchargée et l'échec est notifié à l'application appelante. De même, si une exception non gérée se produit durant l'exécution du code d'initialisation, l'application appelante est notifiée de l'échec du chargement de la bibliothèque.

Voici un exemple de bibliothèque utilisant du code d'initialisation et une procédure de sortie.

```
library Test;

var
  SaveExit: Pointer;

procedure LibExit;
begin
  ⋮  // code de sortie de la bibliothèque
  ExitProc := SaveExit;  // rétablit la chaîne des procédures de sortie
end;
```

```
begin
  ⋮ // code d'initialisation de la bibliothèque
  SaveExit := ExitProc;  // mémoriser la chaîne des procédures de sortie
  ExitProc := @LibExit;  // installer la procédure de sortie LibExit
end.
```

Quand une bibliothèque est déchargée, sa procédure de sortie est exécutée par des appels répétés de l'adresse stockée dans *ExitProc*, jusqu'à ce *ExitProc* contienne la valeur **nil**. La partie initialisation de toutes les unités utilisées par une bibliothèque est exécutée avant le code d'initialisation de la bibliothèque ; la partie finalisation de ces unités est exécutée après la procédure de sortie de la bibliothèque.

Variables globales d'une bibliothèque

Les variables globales déclarées dans une bibliothèque partagée ne peuvent être importées par une application Pascal Objet.

Une bibliothèque peut être utilisée simultanément par plusieurs applications, mais chaque application possède une copie de la bibliothèque dans son propre espace de processus avec son propre jeu de variables globales. Pour que plusieurs bibliothèques, ou plusieurs instances d'une bibliothèque, partagent de la mémoire, elles doivent utiliser des fichiers projetés en mémoire. Pour davantage d'informations, voir votre documentation système.

Bibliothèques et variables système

Plusieurs variables déclarées dans l'unité *System* sont utiles pour ceux qui programment des bibliothèques. Utilisez *IsLibrary* pour déterminer si le code est exécuté dans une application ou dans une bibliothèque ; *IsLibrary* renvoie toujours *False* dans une application et *True* dans une bibliothèque. Pendant la durée de vie une bibliothèque, *HInstance* contient son handle d'instance. *CmdLine* vaut toujours **nil** pour une bibliothèque.

La variable *DLLProc* permet à une bibliothèque de surveiller les appels effectués par le système d'exploitation au point d'entrée de la bibliothèque. Cette caractéristique n'est normalement utilisée que par les bibliothèques qui gèrent le fonctionnement multithread. *DLLProc* est disponible à la fois sous Windows et sous Linux, mais son usage est différent. Sous Windows, *DLLProc* est utilisée dans les multithread applications multithread ; sous Linux, elle sert à déterminer quand votre bibliothèque est en train d'être déchargée. Vous devez utiliser les sections de finalisation, plutôt que les procédures de sortie, pour gérer le comportement d'arrêt. (Voir “Section finalisation” à la page 3-5.)

Pour surveiller les appels du système d'exploitation, créez une procédure callback ne prenant qu'un seul paramètre entier, par exemple :

```
procedure DLLHandler(Reason: Integer);
```

et affectez l'adresse de cette procédure à la variable *DLLProc*. Quand la procédure est appelée, le système lui transmet l'une des valeurs suivantes :

DLL_PROCESS_DETACH	Indique que la bibliothèque se détache de l'espace d'adresse du processus appelant à cause d'une sortie normale ou de l'appel de *FreeLibrary* (ou *dlclose* sous Linux).
DLL_THREAD_ATTACH	Indique que le processus en cours est en train de créer un nouveau thread (Windows seulement).
DLL_THREAD_DETACH	Indique la sortie normale d'un thread (Windows seulement).

Sous Linux, elles sont définies dans l'unité *Libc*.

Dans le corps de la procédure, vous pouvez spécifier les actions à effectuer en fonction du paramètre transmis à la procédure.

Exceptions et erreurs d'exécution dans les bibliothèques

Quand une exception est déclenchée dans une bibliothèque à chargement dynamique sans être gérée, elle se propage depuis la bibliothèque vers le programme appelant. Si l'application ou la bibliothèque appelante est écrite en Pascal Objet, l'exception peut être gérée dans une instruction **try...except** normale. Si l'application ou l'objet partagé appelant est écrit dans un autre langage, l'exception peut être gérée par une exception du système d'exploitation de code d'exception $0EEDFACE. La première entrée du tableau *ExceptionInformation* de l'enregistrement de l'exception du système d'exploitation contient l'adresse de l'exception et la deuxième entrée contient une référence à l'objet exception Pascal Objet.

Généralement, vous ne devez pas laisser des exceptions s'échapper de votre bibliothèque. Sous Windows, les exceptions Delphi correspondent au modèle d'exceptions du système d'exploitation ; Linux ne dispose pas de modèle d'exceptions.

Si une bibliothèque n'utilise pas l'unité *SysUtils*, la gestion des exceptions est désactivée. Dans ce cas, quand une erreur d'exécution se produit dans la bibliothèque, l'application appelante s'arrête. Comme la bibliothèque ne peut pas savoir si elle a été appelée par un programme Pascal Objet, elle ne peut déclencher les procédures de sortie de l'application ; l'application est simplement arrêtée et retirée de la mémoire.

Gestionnaire de mémoire partagée (Windows seulement)

Sous Windows, si une DLL exporte des routines qui transmettent des chaînes longues ou des tableaux dynamiques comme paramètres ou comme résultat de fonction, que ce soit directement ou à l'intérieur d'enregistrements ou d'objets, la DLL et ses applications client (ou DLL) doivent toutes utiliser l'unité *ShareMem*. Cela s'applique également si une application ou une DLL alloue avec *New* ou *GetMem* de la mémoire qui est désallouée par l'appel de *Dispose* ou *FreeMem* dans un autre module. Quand elle est utilisée, l'unité *ShareMem* doit toujours

être la première unité énumérée dans la clause **uses** du programme ou de la bibliothèque l'utilisant.

ShareMem est l'unité d'interface du gestionnaire de mémoire BORLANDMM.DLL qui permet à des modules de partager de la mémoire allouée dynamiquement. BORLANDMM.DLL doit être déployé avec les applications et les DLL qui utilisent l'unité *ShareMem*. Quand une application ou une DLL utilise *ShareMem*, son gestionnaire de mémoire est remplacé par celui contenu dans BORLANDMM.DLL..

Linux utilise *malloc* de glibc pour gérer la mémoire partagée.

Paquets

Un paquet est une bibliothèque compilée spécialement et utilisée par des applications, par l'EDI ou par les deux. Les paquets vous permettent de réorganiser l'emplacement du code sans affecter le code source. On appelle parfois cela le *partitionnement d'application*.

Les paquets d'exécution servent quand un utilisateur exécute une application. *Les paquets de conception* sont utilisés pour installer des composants dans l'EDI et pour créer les éditeurs de propriétés spéciaux de composants personnalisés. Un même paquet peut fonctionner simultanément comme paquet de conception et comme paquet d'exécution ; les paquets de conception font fréquemment référence à des paquets d'exécution dans leur clause **requires**.

Pour les distinguer des autres bibliothèques, les paquets sont stockés dans des fichiers :

- Sous Windows, les noms des fichiers paquet ont l'extension .bpl (Borland package library).
- Sous Linux, les paquets commencent généralement par le préfixe bpl et ont pour extension .so.

Généralement, les paquets sont chargés de manière statique au démarrage d'une application. Vous pouvez cependant utiliser les routines *LoadPackage* et *UnloadPackage* (définies dans l'unité *SysUtils*) pour charger des paquets dynamiquement.

Remarque Quand une application utilise des paquets, le nom de chaque unité empaquetée doit quand même apparaître dans la clause **uses** de tous les fichiers source qui y font référence. Pour davantage d'informations sur les paquets, voir l'aide en ligne.

Déclarations et fichiers source de paquets

Chaque paquet est déclaré dans un fichier source séparé qui doit être enregistré sous l'extension .dpk pour éviter la confusion avec d'autres types de fichier contenant du code Pascal Objet. Le fichier source d'un paquet ne contient pas de

déclarations de type, de données, de procédure ou de fonction. Il contient à la place :

- Le *nom* du paquet.
- Une liste des autres paquets nécessaires (*requires*) au nouveau paquet. Le nouveau paquet est lié à ces paquets.
- Une liste des fichiers unité contenus (*contains*) par ou liés dans le paquet lors de sa compilation. Un paquet sert essentiellement d'enveloppe à ces unités de code source qui fournissent les fonctionnalités du paquet compilé.

La déclaration d'un paquet a la forme suivante :

```
package nomPaquet;
  clauseRequires;
  clauseContains;
end.
```

où *nomPaquet* est un identificateur valide. *clauseRequires et clauseContains* sont tous les deux facultatifs. Par exemple, le code suivant déclare le paquet *DATAX* :

```
package DATAX;
requires
  baseclx,
  visualclx;
  contains Db, DBLocal, DBXpress, ... ;
end.
```

La clause **requires** énumère les autres paquets externes utilisés par le paquet qui est déclaré. Elle est constituée de la directive **requires** suivie d'une liste, délimitée par des virgules, de noms de paquet, suivie d'un point-virgule. Si un paquet ne fait référence à aucun autre paquet, il n'a pas besoin de la clause **requires**.

La clause **contains** identifie les fichiers unité qui sont compilés et associés dans le paquet. Elle est constituée de la directive **contains**, suivie d'une liste délimitée par des virgules de noms d'unité, suivie d'un point-virgule. Chaque nom d'unité peut être suivi du mot réservé **in** et du nom, entre apostrophes, d'un fichier source avec ou sans chemin d'accès. Le chemin d'accès peut être relatif ou absolu. Par exemple :

```
contains MonUnit in 'C:\MonProjet\MonUnit.pas';
```

Remarque Les variables locales thread (déclarées avec **threadvar**) d'une unité paquetée ne peuvent être accessibles aux clients qui utilisent le paquet.

Nom de paquets

La compilation d'un paquet génère plusieurs fichiers. Par exemple, le fichier source du paquet *DATAX* est le fichier DATAX.dpk, sa compilation génère un exécutable et une image binaire appelés

- Sous Windows : DATAX.bpl et DATAX.dcp
- Sous Linux: bplDATAX.so (Linux) et DATAX.dcp.

DATAX est utilisé pour faire référence au paquet dans la clause **requires** d'autres paquets ou pour utiliser le paquet dans une application. Les noms de paquet doivent être uniques à l'intérieur d'un projet.

Clause requires

La clause **requires** liste les autres paquets, externes, utilisés par le paquet en cours. Elle fonctionne comme la clause **uses** d'un fichier unité. A la compilation, un paquet externe énuméré dans la clause **requires** est automatiquement lié dans une application utilisant le paquet en cours et l'une des unités contenues dans le paquet externe.

Si les fichiers unité contenus dans un paquet font des références à d'autres unités empaquetées, les autres paquets doivent être inclus dans la clause **requires** du premier paquet. Si d'autres paquets sont omis dans la clause **requires**, le compilateur charge les unités référencées à partir des fichiers .dcu (Windows) ou .dpu (Linux).

Références de paquet circulaires

Un paquet ne peut contenir de références circulaires dans sa clause **requires**. Cela signifie :

- Qu'un paquet ne peut se référencer lui-même dans sa propre clause **requires**.
- Qu'une chaîne de références doit se terminer sans référencer un paquet déjà référencé dans la chaîne. Si le paquet *A* nécessite le paquet *B*, alors le paquet *B* ne peut nécessiter le paquet *A. De même, si le paquet A* nécessite le *B* et si le paquet *B* nécessite le paquet *C*, alors le paquet *C* ne peut nécessiter le paquet *A*.

Références de paquet répétées

Le compilateur ne tient pas compte des références de paquet répétées dans la clause **requires** d'un paquet. Néanmoins, dans un souci de clarté et de lisibilité des programmes, il est souhaitable de supprimer les références dupliquées.

Clause contains

La clause **contains** identifie les fichiers unités qui sont liés dans le paquet. Ne spécifiez pas l'extension de nom de fichier dans la clause **contains**.

Utilisation redondante de code source

Un paquet ne peut apparaître dans la clause **contains** d'un autre paquet, ni dans la clause **uses** d'une unité.

Toutes les unités indiquées directement dans la clause **contains** d'un paquet, ou indirectement dans la clause **uses** de ces unités sont liées dans le paquet lors de la compilation. Les unités contenues (directement ou non) dans un paquet ne peuvent pas être incluses dans d'autres paquets référencés dans la clause **requires** de ce paquet.

Une unité ne peut pas être contenue (directement ou non) dans plus d'un des paquets utilisés par une même application.

Compilation de paquets

Les paquets sont généralement compilés dans l'EDI en utilisant les fichiers .dpk générés par l'éditeur de paquet. Vous pouvez également compiler des fichiers .dpk directement depuis la ligne de commande. Quand vous générez un projet contenant un paquet, le paquet est automatiquement recompilé, si c'est nécessaire.

Fichiers générés

Le tableau suivant indique les fichiers générés par la compilation réussie d'un paquet :

Tableau 9.1 Fichiers d'un paquet compilé

Extension de fichier	Contenu
dcp	Une image binaire contenant l'en-tête du paquet et la concaténation de tous les fichiers dcu (Windows) ou dpu (Linux) du paquet. Un seul fichier dcp est créé par paquet. Le nom de base du fichier dcp est celui du fichier source dpk.
dcu (Windows) dpu (Linux)	L'image binaire d'un fichier unité contenu dans un paquet. Un fichier dcu est créé, si nécessaire, pour chaque fichier unité.
bpl<paquet>.so sous Linux	Le paquet d'exécution. Ce fichier est une bibliothèque partagée ayant des fonctionnalités propres à Borland. Le nom de base du paquet est celui du fichier source dpk.

Plusieurs directives de compilation et options de la ligne de commande permettent de gérer la compilation des paquets.

Directives de compilation spécifiques aux paquets

Le tableau suivant énumère les directives de compilation spécifiques aux paquets qui peuvent être insérées dans le code source. Pour davantage d'informations, voir l'aide en ligne.

Tableau 9.2 Directives de compilation spécifiques aux paquets

Directive	Fonction
{$IMPLICITBUILD OFF}	Empêche un paquet d'être implicitement recompilé ultérieurement. Utilisez-la dans les fichiers .dpk pour des paquets qui fournissent des fonctionnalités de bas niveau, qui changent rarement entre deux compilations ou dont le code source ne sera pas distribué.
{$G–} or {IMPORTEDDATA OFF}	Désactive la création de références aux données importées. Cette directive augmente les performances d'accès à la mémoire, mais empêche l'unité dans laquelle elle apparaît d'être référencée dans d'autres paquets.

Tableau 9.2 Directives de compilation spécifiques aux paquets

Directive	Fonction
{$WEAKPACKAGEUNIT ON}	Les unités sont “faiblement empaquetées”, comme expliqué dans l’aide en ligne.
{$DENYPACKAGEUNIT ON}	Empêche l’incorporation de l’unité dans un paquet.
{$DESIGNONLY ON}	Compile le paquet pour l’installation dans l’EDI. Cette directive se place dans un fichier .dpk.
{$RUNONLY ON}	Compile un paquet à l’exécution seulement. Cette directive se place dans un fichier .dpk.

L’inclusion de la directive **{$DENYPACKAGEUNIT ON}** dans le code source, empêche le fichier de l’unité d’être empaqueté. L’inclusion de **{$G–}** ou de {**IMPORTEDDATA OFF}** peut empêcher l’utilisation d’un paquet dans une application qui utilise d’autres paquets.

Il est possible d’inclure les autres directives de compilation, quand c’est nécessaire, dans le code source d’un paquet.

Options du compilateur ligne de commande spécifiques aux paquets

Les options suivantes spécifiques aux paquets sont utilisables avec le compilateur en ligne de commande. Pour davantage d’informations, voir l’aide en ligne.

Tableau 9.3 Options du compilateur ligne de commande spécifiques aux paquets

Option	Fonction
–$G–	Neutralise la création des références de données importées. L’utilisation de cette option améliore l’efficacité de l’utilisation de la mémoire, mais empêche les paquets compilés, en l’utilisant, de faire référence aux variables définies dans d’autres paquets.
–LE *chemin*	Spécifie le répertoire où placer le fichier paquet compilé.
–LN *chemin*	Spécifie le répertoire où placer le fichier paquet dcp.
–LU*nomPaquet* *[;nomPaquet2;...]*	Spécifie d’autres paquets d’exécution à utiliser dans une application. S’utilise pour la compilation d’un projet.
–Z	Empêche la recompilation ultérieure implicite d’un paquet. S’utilise pour des paquets qui fournissent des fonctionnalités de bas niveau, qui changent rarement entre deux compilations ou dont le code source ne sera pas distribué.

L’utilisation de l’option **–$G–** peut empêcher l’utilisation d’un paquet dans une application qui utilise d’autres paquets.

Il est possible d’inclure les autres options de compilation, quand c’est nécessaire, pour compiler un paquet.

Chapitre

10

Interfaces d'objets

Une *interface d'objet*, ou plus simplement une *interface*, définit des méthodes qui peuvent être implémentées par une classe. Les interfaces sont déclarées comme des classes mais elles ne peuvent être instanciées directement et n'ont pas leurs propres définitions de méthode. C'est aux classes gérant une interface qu'est laissée la responsabilité de fournir l'implémentation des méthodes de l'interface. Une variable de type interface peut référencer un objet dont la classe implémente cette interface. Cependant, en utilisant cette variable, il n'est possible d'appeler que les méthodes déclarées dans l'interface.

Les interfaces proposent certains des avantages de l'héritage multiple sans en avoir la complexité sémantique. Elles jouent également un rôle essentiel dans l'utilisation des modèles d'objets distribués. Les objets personnalisés qui supportent les interfaces peuvent interagir avec des objets écrits en C++, en Java et dans d'autres langages.

Types interface

Comme les classes, les interfaces ne peuvent se déclarer que dans la portée la plus extérieure d'un programme ou d'une unité, mais pas dans une procédure ou une fonction. La déclaration d'un type interface a la forme suivante :

```
type nomInterface = interface (interfaceAncêtre)
  ['{GUID}']
  listeMembres
end;
```

où (*interfaceAncêtre*) et ['{*GUID*}'] sont facultatifs. Par bien des côtés, la déclaration d'une interface ressemble à celle d'une classe, mais certaines restrictions s'appliquent :

- La *listeMembres* ne peut contenir que des méthodes et des propriétés ; les champs ne sont pas autorisés dans les interfaces.

- Une interface n'ayant pas de champ, les spécificateurs de propriété **read** et **write** doivent être des méthodes.
- Tous les membres d'une interface sont publics. Les spécificateurs de visibilité et de stockage ne sont pas autorisés (il est par contre possible d'utiliser **default** avec une propriété tableau).
- Les interfaces n'ont ni constructeurs, ni destructeurs. Elles ne peuvent être instanciées, si ce n'est via des classes qui implémentent leurs méthodes.
- Il n'est pas possible de déclarer des méthodes comme étant **virtual**, **dynamic**, **abstract** ou **override**. Comme les interfaces n'implémentent pas leur propres méthodes, ces directives sont dépourvues de sens.

Voici un exemple de déclaration d'interface :

```
type
  IMalloc = interface(IInterface)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
  end;
```

Dans certaines déclarations d'interfaces, le mot réservé **interface** est remplacé par **dispinterface**. Cette construction (avec les directives **dispid**, **readonly** et **writeonly**) est spécifique à la plate-forme et n'est pas utilisée dans la programmation Linux.

IInterface et l'héritage

Comme une classe, une interface hérite de toutes les méthodes de ces ancêtres. Mais les interfaces, à la différence des classes, *n'implémentent pas* les méthodes. Ce dont hérite l'interface, c'est de *l'obligation* d'implémenter les méthodes ; obligation qui est en fait supportée par toute classe gérant l'interface.

La déclaration d'une interface peut spécifier une interface ancêtre. Si l'ancêtre n'est pas spécifié, l'interface descend directement de *IInterface*, qui est définie dans l'unité *System* et qui est l'ancêtre ultime de toutes les autres interfaces. *IUnknown* déclare trois méthodes : *QueryInterface*, *_AddRef* et *_Release*.

Remarque *IInterface* est équivalent à *IUnknown.* Vous devez généralement utiliser *IInterface* pour les applications indépendantes des plates-formes et réserver l'usage de *IUnknown* au programmes spécifiques qui incluent des dépendances Windows.

QueryInterface permet de parcourir librement les différentes interfaces gérées par un objet. *_AddRef* et *_Release* proposent la gestion de la durée de vie des références d'interface. Le moyen le plus simple d'implémenter ces méthodes consiste à dériver la classe d'implémentation à partir de la classe *TInterfacedObject* définie dans l'unité *System*. Il est également possible de se passer de l'une quelconque de ces méthodes en l'implémentant en tant que fonction vide ;

cependant, les objets COM (Windows seulement), doivent être gérés via *_AddRef* et *_Release*

Identification d'interface

La déclaration d'une interface peut spécifier un identificateur globalement unique (GUID) représenté par un littéral chaîne placé entre crochets juste avant la liste des membres. La partie GUID de la déclaration doit avoir la forme suivante :

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

où chaque *x* est un chiffre hexadécimal (de 0 à 9 et de A à F). Sous Windows, l'éditeur de bibliothèque de types génère automatiquement des GUID pour les nouvelles interfaces ; vous pouvez également générer des GUID en appuyant sur *Ctrl+Maj+G* dans l'éditeur de code (sous Linux, vous devez utiliser *Ctrl+Maj+G*).

Un GUID est une valeur binaire sur 16 octets qui identifie une interface de manière unique. Si une interface a un GUID, vous pouvez utiliser l'interrogation d'interface pour obtenir des références à ses implémentations (voir "Interrogation d'interface" à la page 10-11).

Les types *TGUID* et *PGUID*, déclarés de la manière suivante dans l'unité *System*, sont utilisés pour manipuler des GUID :

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

Quand vous spécifiez une constante typée de type *TGUID*, vous pouvez utiliser un littéral chaîne pour spécifier sa valeur. Par exemple :

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

Dans des appels de procédure ou de fonction, un GUID ou un identificateur d'interface peut s'utiliser comme paramètre, par valeur ou constante, de type *TGUID*. Par exemple, étant donné la déclaration suivante :

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports peut être appelée de deux manières :

```
if Supports(Allocator, IMalloc) then ...
if Supports(Allocator, IID_IMalloc) then ...
```

Conventions d'appel des interfaces

La convention d'appel par défaut est **register** mais les interfaces partagées par plusieurs modules (en particulier si ces modules sont écrits avec des langages différents) doivent déclarer toutes leurs méthodes avec la convention **stdcall**.

Utilisez **safecall** pour implémenter les interfaces CORBA. Sous Windows, vous pouvez utiliser **safecall** pour implémenter les méthodes des interfaces doubles (comme décrit dans “Interfaces doubles (Windows seulement)” à la page 10-14).

Pour davantage d’informations sur les conventions d’appel, voir “Conventions d’appel” à la page 6-5.

Propriétés d’interface

Les propriétés déclarées dans une interface ne sont accessibles que dans des expressions de type interface : il n’est pas possible d’y accéder via des variables de type classe. De plus, les propriétés d’interface ne sont visibles que dans les programmes où l’interface est compilée. Par exemple, sous Windows, les objets COM n’ont pas de propriété.

Dans une interface, les spécificateurs **read** et **write** d’une propriété doivent être des méthodes puisque les champs ne sont pas utilisables.

Déclarations avancées

La déclaration d’une interface qui se termine avec le mot réservé **interface** et un point-virgule sans spécifier l’ancêtre, le GUID et la liste des membres sont des *déclarations avancées*. Une déclaration avancée doit être complétée par une *déclaration de définition* de la même interface à l’intérieur de la même section de déclaration de types. En d’autres termes, entre une déclaration avancée et sa déclaration de définition, il ne peut y avoir que d’autres déclarations de type.

Les déclarations avancées permettent de déclarer des interfaces mutuellement dépendantes. Par exemple :

```
type
  IControl = interface;
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000044}']
    function GetControl(Index: Integer): IControl;
    ⋮
  end;
  IControl = interface
    ['{00000115-0000-0000-C000-000000000049}']
    function GetWindow: IWindow;
    ⋮
  end;
```

Par contre, les interfaces dérivées mutuellement dépendantes ne sont pas autorisées. Ainsi, il n’est pas permis de dériver *IWindow* de *IControl* et de dériver également *IControl* de *IWindow*.

Implémentation des interfaces

Une fois une interface déclarée, elle doit être implémentée par une classe avant de pouvoir être utilisée. Les interfaces implémentées par une classe sont spécifiées dans la déclaration de la classe après le nom de l'ancêtre de la classe. De telles déclarations ont la forme :

```
type nomClasse = class (classeAncêtre, interface1, ..., interfacen)
  listeMembres
 end;
```

Par exemple :

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    ⋮
  end;
```

déclare une classe appelée *TMemoryManager* qui implémente les interfaces *IMalloc* et *IErrorInfo*. Quand une classe implémente une interface, elle doit implémenter (ou hériter d'une implémentation) de chaque méthode déclarée dans l'interface.

Voici la déclaration de *TInterfacedObject* dans l'unité *System* :

```
type
TInterfacedObject = class(TObject, IInterface)
protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

TInterfacedObject implémente l'interface *IInterface,* donc *TInterfacedObject* déclare et implémente chacune des trois méthodes de *IInterface*.

Les classes qui implémentent des interfaces peuvent également s'utiliser comme classes de base (le premier exemple ci-dessus déclare *TMemoryManager* comme descendant direct de *TInterfacedObject*). Comme chaque interface hérite de *IInterface*, une classe qui implémente des interfaces doit implémenter les méthodes *QueryInterface*, *_AddRef* et *_Release*. L'interface *TInterfacedObject* de l'unité *System* implémente ces méthodes et fournit donc une base pratique pour dériver des classes qui implémentent des interfaces.

Quand une interface est implémentée, chacune de ses méthodes est mise en correspondance avec une méthode de la classe d'implémentation ayant le même type de résultat, la même convention d'appel, le même nombre de paramètres et ayant à chaque position le même type de paramètre. Par défaut, chaque méthode de l'interface est associée à la méthode portant le même nom dans la classe d'implémentation.

Clauses de résolution de méthode

Vous pouvez redéfinir le mécanisme par défaut d'association basé sur le nom en spécifiant des *clauses de résolution de méthode* dans la déclaration d'une classe. Quand une classe implémente plusieurs interfaces ayant des méthodes portant le même nom, les clauses de résolution de méthode vous permettent d'éviter les conflits de nom.

Une clause de résolution de méthode a la forme suivante :

procedure *interface*.*méthodeInterface* = *méthodeImplémentant*;

ou la suivante :

function *interface*.*méthodeInterface* = *méthodeImplémentant*;

où *méthodeImplémentant* est une méthode déclarée dans la classe ou dans l'un de ses ancêtres. La *méthodeImplémentant* peut être une méthode déclarée dans la déclaration de classe, mais ce ne peut pas être une méthode privée d'une classe ancêtre déclarée dans un autre module.

Par exemple, la déclaration de classe suivante :

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ⋮
  end;
```

associe les méthodes *Alloc* et *Free* de *IMalloc* aux méthodes *Allocate* et *Deallocate de TMemoryManager*.

Une clause de résolution de méthode ne peut changer l'association introduite dans une classe ancêtre.

Modification de l'implémentation héritée

Les classes dérivées peuvent changer la manière dont est implémentée une méthode d'interface spécifique en surchargeant la méthode d'implémentation. Il est nécessaire pour ce faire que la méthode d'implémentation soit virtuelle ou dynamique.

Une classe peut également réimplémenter la totalité d'une interface qu'elle hérite d'une classe ancêtre. Cela nécessite de spécifier à nouveau l'interface dans la déclaration de la classe dérivée. Par exemple :

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    ⋮
  end;

  TWindow = class(TInterfacedObject, IWindow)  // TWindow implémente IWindow
```

```
    procedure Draw;
    ⋮
  end;

  TFrameWindow = class(TWindow, IWindow)  // TFrameWindow ré-implemente IWindow
    procedure Draw;
    ⋮
  end;
```

La réimplémentation d'une interface masque l'implémentation héritée de la même interface. Dans ce cas, les clauses de résolution de méthode d'une classe ancêtre ne s'appliquent pas à l'interface réimplémentée.

Implémentation des interfaces par délégation

La directive **implements** vous permet de déléguer l'implémentation d'une interface à une propriété de la classe d'implémentation. Par exemple :

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

déclare une propriété appelée *MyInterface* qui implémente l'interface *IMyInterface*.

La directive **implements** doit être le dernier spécificateur dans la déclaration de la propriété, elle peut énumérer plusieurs interfaces, séparées par des virgules. La propriété déléguée :

- Doit être de type classe ou interface.
- Ce ne peut pas être une propriété tableau et elle ne peut pas avoir de spécificateur d'indice.
- Elle doit avoir un spécificateur **read**. Si la propriété utilise une méthode **read**, cette méthode doit utiliser la convention d'appel par défaut **register**, elle ne peut pas être dynamique (par contre, elle peut être virtuelle) et elle ne peut pas spécifier la directive **message**.

Remarque La classe que vous utilisez pour implémenter l'interface déléguée doit dériver de *TAggregatedObject*.

Délégation à une propriété de type interface

Si la propriété déléguée est de type interface, cette interface (ou une interface dont elle dérive) doit apparaître dans la liste des ancêtres de la classe dans laquelle la propriété est déclarée. La propriété déléguée doit renvoyer un objet dont la classe implémente complètement l'interface spécifiée par la directive **implements** et ce sans utiliser de clauses de résolution de méthode. Par exemple :

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

  TMyClass = class(TObject, IMyInterface)
    FMyInterface: IMyInterface;
```

```
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyInterface := ...  // un objet dont la classe implémente IMyInterface
  MyInterface := MyClass;
  MyInterface.P1;
end;
```

Délégation à une propriété de type classe

Si la propriété déléguée est de type classe, les méthodes implémentant l'interface spécifiée sont d'abord recherchées dans la classe et ses ancêtres, puis dans la classe conteneur et dans ses ancêtres. Il est ainsi possible d'implémenter certaines méthodes dans la classe spécifiée par la propriété et d'autres dans la classe dans laquelle la propriété est déclarée. Il est possible d'utiliser de manière normale les clauses de résolution de méthode afin de résoudre les ambiguïtés ou de spécifier une méthode particulière. Une interface ne peut être implémentée par plus d'une propriété de type classe. Par exemple :

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;
  TMyImplClass = class
    procedure P1;
    procedure P2;
  end;
  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyImplClass: TMyImplClass;
    property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
  end;
procedure TMyImplClass.P1;
  ⋮
procedure TMyImplClass.P2;
  ⋮
procedure TMyClass.MyP1;
  ⋮
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyImplClass := TMyImplClass.Create;
  MyInterface := MyClass;
  MyInterface.P1;        // appelle TMyClass.MyP1;
  MyInterface.P2;        // appelle TImplClass.P2;
end;
```

Références d'interface

Si vous déclarez une variable de type interface, la variable peut référencer des instances de toute classe qui implémentent l'interface. De telles variables vous permettent d'appeler les méthodes de l'interface sans savoir, au moment de la compilation, où l'interface est implémentée. Leur utilisation est sujette aux limitations suivantes :

- Une expression de type interface ne vous donne accès qu'aux méthodes et propriétés déclarées dans l'interface et pas aux autres membres de la classe d'implémentation.
- Une expression de type interface ne peut référencer un objet dont la classe implémente une interface dérivée sauf si la classe (ou une classe qui en dérive implémente aussi explicitement l'interface ancêtre.

Par exemple :

```
type
  IAncestor = interface
  end;

  IDescendant = interface(IAncestor)
    procedure P1;
  end;

  TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
    procedure P2;
  end;
  ⋮
var
  D: IDescendant;
  A: IAncestor;
begin
  D := TSomething.Create;  // Correct!
  A := TSomething.Create;  // erreur
  D.P1;  // Correct!
  D.P2;  // erreur
end;
```

Dans cet exemple :

- *A* est déclarée comme variable de type *IAncestor*. Comme *TSomething* n'énumère pas *IAncestor* dans les interfaces qu'elle implémente, une instance de *TSomething* ne peut pas être affectée à *A*. Mais, si la déclaration de *TSomething* est changée en :

  ```
  TSomething = class(TInterfacedObject, IAncestor, IDescendant)
    ⋮
  ```

  la première erreur devient une affectation légale.
- *D* est déclarée comme variable de type *IDescendant*. Quand *D* désigne une instance de *TSomething*, il n'est pas possible de l'utiliser pour accéder à la

méthode *P2* de *TSomething*, car *P2* n'est pas une méthode de *IDescendant*. Mais, si la déclaration de *D* est changée en :

```
D: TSomething;
```

la deuxième erreur devient un appel de méthode autorisé.

Les références d'interface sont gérées par un système de comptage de références qui dépend des méthodes *_AddRef* et *_Release* héritées de *IInterface*. Quand un objet n'est référencé que par l'intermédiaire d'interfaces, il n'est pas nécessaire de le détruire manuellement : l'objet est détruit automatiquement quand sa dernière référence sort de portée.

La seule initialisation possible pour les variables globales de type interface est **nil**.

Pour déterminer à quel moment une expression de type interface désigne un objet, il faut la transmettre à la fonction standard *Assigned*.

Compatibilité des affectations d'interfaces

Un type classe est compatible pour l'affectation avec tous les types interface implémentés par la classe. Un type interface est compatible pour l'affectation avec tous ses types interface ancêtre. Il est possible d'affecter la valeur **nil** à toute variable de type interface.

Il est possible d'affecter une expression de type interface à un variant. Si l'interface est de type *IDispatch* ou d'un type descendant, le variant reçoit le code de type *varDispatch* ; sinon le variant reçoit le code de type *varUnknown*.

Un variant dont le code de type est *varEmpty*, *varUnknown* ou *varDispatch* peut être affecté à une variable *IInterface*. Un variant dont le type de code est *varEmpty* ou *varDispatch* peut être affecté à une variable *IDispatch*.

Transtypage d'interfaces

Les types interface respectent les mêmes règles que les types classe pour les transtypages de variables et de valeurs. Les expressions de type classe peuvent être transtypées en types interface (par exemple, `IMyInterface(SomeObject)`) dans la mesure où la classe implémente l'interface.

Une expression de type interface peut être transtypée en *Variant*. Si l'interface est de type *IDispatch* ou d'un type descendant, le variant résultant a le code de type *varDispatch* ; sinon, le variant résultant a le code de type *varUnknown*.

Un variant dont le code de type est *varEmpty*, *varUnknown* ou *varDispatch* peut être transtypé en *IInterface*. Un variant dont le code de type est *varEmpty* ou *varDispatch* peut être transtypé en *IDispatch*.

Interrogation d'interface

Vous pouvez utiliser l'opérateur **as** afin d'effectuer des transtypages d'interface avec vérification. Ce mécanisme s'appelle *l'interrogation d'interfaces* ; il produit une expression de type interface depuis une référence d'objet ou une autre référence d'interface à partir du type réel (à l'exécution) de l'objet. Une interrogation d'interface a la forme suivante :

objet `as` *interface*

où *objet* est une expression de type interface, de type variant ou désignant une instance d'une classe qui implémente une interface, et où *interface* est une interface déclarée avec un GUID.

L'interrogation d'interface renvoie **nil** si *objet* vaut **nil**. Sinon, elle transmet le GUID de *interface* à la méthode *QueryInterface* de *objet* et déclenche une exception sauf si *QueryInterface* renvoie zéro. Si *QueryInterface* renvoie zéro (ce qui indique que la classe de *objet* implémente *interface*), l'interrogation d'interface renvoie une référence sur *objet*.

Objets automation (Windows seulement)

Un objet dont la classe implémente l'interface *IDispatch* (déclarée dans l'unité *System*) est appelé un objet automation. L'automation est disponible uniquement sous Windows.

Types interface de répartition (Windows seulement)

Les types interface de répartition définissent les méthodes et propriétés qu'un objet automation implémente via *IDispatch*. Les appels à une interface de répartition sont routés à l'exécution par la méthode *Invoke de IDispatch*. Une classe ne peut pas implémenter une interface de répartition.

La déclaration d'une interface de répartition a la forme suivante :

```
type nomInterface = dispinterface
  ['{GUID}']
  listeMembres
end;
```

où `['{`*GUID*`}']` est facultatif et *listeMembres* est constitué de déclarations de propriétés et de méthodes. La déclaration d'une interface de répartition est semblable à celle d'une interface normale, mais elle ne peut pas spécifier d'ancêtre. Par exemple :

```
type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
```

```
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
  end;
```

Méthodes des interfaces de répartition (Windows seulement)

Les méthodes des interfaces de répartition sont le prototype d'appels de la méthode *Invoke* de l'implémentation *IDispatch* sous-jacente. Pour spécifier l'identificateur de répartition automation d'une méthode, utilisez la directive **dispid** dans sa déclaration en la faisant suivre d'une constante entière ; la spécification d'un identificateur déjà utilisé provoque une erreur.

Une méthode déclarée dans une interface de répartition ne peut contenir d'autres directives que **dispid**. Le type des paramètres et du résultat doit être un type automatisable : *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool* ou tout type interface.

Propriétés des interfaces de répartition

Les propriétés des interfaces de répartition ne doivent pas indiquer de spécificateurs d'accès. Elles peuvent être déclarées comme **readonly** ou **writeonly**. Pour spécifier l'identificateur de répartition d'une propriété, spécifiez dans la déclaration de la propriété la directive **dispid** suivie d'une constante entière. La spécification d'un identificateur déjà utilisé provoque une erreur. Les propriétés tableau peuvent être déclarées comme **default**. Toutes les autres directives sont interdites dans la déclaration des propriétés des interfaces de répartition.

Accès aux objets automation (Windows seulement)

Utilisez des variants pour accéder aux objets automation. Quand un variant référence un objet automation, vous pouvez appeler les méthodes de l'objet et lire ou modifier ses propriétés via le variant. Pour ce faire, vous devez inclure *ComObj* dans la clause **uses** de votre unité, de votre programme ou de votre bibliothèque.

Les appels des méthodes automation sont reliés uniquement à l'exécution et ne nécessitent pas de déclaration de méthode préalable. La validité de ces appels n'est donc pas vérifiée à la compilation.

L'exemple suivant illustre l'utilisation d'appels de méthodes automation. La fonction *CreateOleObject* (définie dans *ComObj*) renvoie une référence *IDispatch* à un objet automation compatible pour l'affectation avec le variant *Word* :

```
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('Ceci est la première ligne'#13);
```

```
    Word.Insert('Ceci est la seconde ligne'#13);
    Word.FileSaveAs('c:\temp\test.txt', 3);
  end;
```

Il est possible de transmettre des paramètres de type interface aux méthodes automation.

Les tableaux variant dont le type d'élément est *varByte* constituent le meilleur moyen de transmettre des données binaires entre un contrôleur et un serveur automation. En effet, les données de tels tableaux ne sont pas traduites et elles sont accessibles de manière efficace en utilisant les routines *VarArrayLock* et *VarArrayUnlock*.

Syntaxe des appels de méthode d'objet automation

La syntaxe de l'appel d'une méthode d'un objet automation ou de l'accès à l'une de ses propriétés est similaire à la syntaxe normale d'un appel de méthode ou de l'accès à une propriété. Cependant, les appels de méthodes peuvent utiliser des paramètres *par position* et des paramètres *nommés* (certains serveurs automation ne gèrent pas les paramètres nommés).

Un paramètre par position est simplement une expression. Un paramètre nommé est constitué d'un identificateur de paramètre suivi du symbole **:=** et d'une expression. Dans un appel de méthode, les paramètres par position doivent précéder les paramètres nommés. Les paramètres nommés peuvent être spécifiés dans un ordre quelconque.

Certains serveurs automation vous autorisent à omettre des paramètres dans un appel de méthode en utilisant alors des valeurs par défaut. Par exemple :

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Les paramètres d'appel des méthodes automation peuvent être de type entier, réel, chaîne, booléen ou variant. Un paramètre est transmis par adresse si l'expression du paramètre est constituée uniquement d'une référence de variable et si la référence de variable est de type *Byte*, *Smallint*, *Integer*, *Single*, *Double*, *Currency*, *TDateTime*, *AnsiString*, *WordBool* ou *Variant*. Si l'expression n'est pas de l'un de ces types ou si elle n'est pas composée uniquement d'une variable, le paramètre est transmis par valeur. Le transfert d'un paramètre par adresse à une méthode qui attend un paramètre par valeur oblige COM à extraire la valeur du paramètre par adresse. Le transfert d'un paramètre par valeur à une méthode qui attend un paramètre par adresse déclenche une erreur.

Interfaces doubles (Windows seulement)

Une interface double est une interface qui gère à la fois la liaison à la compilation et la liaison à l'exécution en utilisant l'automation. Les interfaces doubles doivent dériver de *IDispatch*.

Toutes les méthodes d'une interface double (à l'exception de celles héritées de *IInterface* et de *IDispatch*) doivent utiliser la convention **safecall** et le type du résultat et des paramètres doit être automatisable. Les types automatisables sont *Byte*, *Currency*, *Real*, *Double*, *Real48*, *Integer*, *Single*, *Smallint*, *AnsiString*, *ShortString*, *TDateTime*, *Variant*, *OleVariant* et *WordBool*.)

Chapitre

11

Gestion de la mémoire

Ce chapitre explique comment les programmes utilisent la mémoire et décrit le format interne des types de données Pascal Objet.

Le gestionnaire de mémoire (Windows seulement)

Remarque Linux utilise des fonctions glibc comme *malloc* pour la gestion de la mémoire. Pour plus d'informations, reportez-vous à la page man *malloc* de votre système Linux.

Dans les systèmes Windows, le gestionnaire de mémoire gère toutes les allocations et désallocations de mémoire dynamique d'une application. Les procédures standard *New*, *Dispose*, *GetMem*, *ReallocMem* et *FreeMem* utilisent le gestionnaire de mémoire, et tous les objets et chaînes longues sont allouées par son intermédiaire.

Sous Windows, le gestionnaire de mémoire est optimisé pour les applications qui allouent une grand nombre de blocs de taille petite à moyenne, comme c'est habituellement le cas pour les applications orientées objet et celles qui traitent des données chaînes. D'autres gestionnaires de mémoire, tels les implémentations de *GlobalAlloc*, *LocalAlloc*, et le support du tas privé de Windows, ne fonctionnent généralement pas bien dans ce cas, et entraînent des chutes de vitesse de l'application s'ils sont utilisés directement.

Pour garantir les meilleures performances, le gestionnaire de mémoire s'interface directement avec l'API de mémoire virtuelle de Win32 (les fonctions *VirtualAlloc* et *VirtualFree*). Le gestionnaire de mémoire réserve la mémoire depuis le système d'exploitation en morceaux de 1 Mo d'espace adresse et affecte la mémoire au fur et à mesure des besoins par incréments de 16 Ko. Il désaffecte et libère la mémoire non utilisée par morceaux de 16 Ko et 1 Mo. Pour les blocs plus petits, la mémoire affectée est sous-allouée plus tard.

Les blocs du gestionnaire de mémoire sont toujours arrondis à une limite supérieure de 4 octets et contiennent toujours un en-tête de 4 octets dans lequel sont stockées la taille du bloc et d'autres bits d'état. Cela veut dire que les blocs de ce gestionnaire de mémoire sont toujours alignés sur un double mot, ce qui garantit des performances CPU optimales lors de l'adressage du bloc.

Le gestionnaire de mémoire utilise deux variables d'état, *AllocMemCount* et *AllocMemSize*, qui contiennent le nombre des blocs mémoire actuellement alloués et la taille combinée de tous les blocs mémoire alloués. Les applications peuvent utiliser ces variables pour afficher des informations d'état du débogage.

L'unité *System* comprend deux procédures, *GetMemoryManager* et *SetMemoryManager*, permettant aux applications d'intercepter les appels de bas niveau du gestionnaire de mémoire. L'unité *System* fournit également une fonction nommée *GetHeapStatus* qui renvoie un enregistrement contenant des informations d'état du gestionnaire de mémoire détaillées. Pour davantage d'informations sur ces routines, voir l'aide en ligne.

Variables

Les variables globales sont allouées dans le segment de données de l'application et persistent pour la durée du programme. Les variables locales (déclarées dans les procédures et les fonctions) sont placées dans la pile de l'application. A chaque appel d'une procédure ou d'une fonction, elle alloue un ensemble de variables locales ; à la sortie, les variables locales sont libérées. Les optimisations du compilateur peuvent libérer des variables encore plus tôt.

Remarque Sous Linux, la talle de la pile est définie uniquement par l'environnement.

Sous Windows, la taille de pile d'une application est définie par deux valeurs : la *taille de pile minimum* et *la taille de pile maximum*. Ces valeurs sont contrôlées par les directives de compilation **$MINSTACKSIZE** et **$MAXSTACKSIZE** qui valent, respectivement, par défaut 16 384 (16 Ko) et 1 048 576 (1 Mo). Une application a la certitude de disposer de la taille de pile minimum et la taille de la pile n'est jamais autorisée à dépasser la taille de pile maximum. S'il n'y a pas assez de mémoire disponible pour la taille de pile minimum d'une application, Windows indique une erreur au démarrage de l'application.

Si une application Windows nécessite davantage d'espace de la pile que ce qui est spécifié par la taille minimum de pile, de la mémoire supplémentaire est allouée par incréments de 4 Ko. Si l'allocation des zones supplémentaires de pile échoue (car il n'y a plus de mémoire disponible ou parce que la taille totale de la pile dépasserait alors la taille maximum), une exception *EStackOverflow* est déclenchée. Les tests de dépassement de la pile sont entièrement automatiques. La directive de compilation **$S** qui contrôlait à l'origine la vérification des dépassements de pile est conservée uniquement pour la compatibilité ascendante.

Sous Windows ou Linux, les variables dynamiques créées avec les procédures *GetMem* et *New* sont allouées sur le tas et persistent tant qu'elles ne sont pas libérées avec *FreeMem* ou *Dispose*.

Les chaînes longues, les chaînes étendues, les tableaux dynamiques, les variants et les interfaces sont alloués sur le tas mais leur mémoire est gérée automatiquement.

Formats de données internes

Les sections suivantes décrivent le format interne des types de données Pascal Objet.

Types entiers

Le format d'une variable de type entier dépend de ses bornes inférieure et supérieure.

- Si les deux bornes sont contenues dans l'intervalle – 128..127 (*Shortint*), la variable est stockée sous la forme d'un octet signé.
- Si les deux bornes sont contenues dans l'intervalle 0..255 (*Byte*), la variable est stockée sous la forme d'un octet non signé.
- Si les deux bornes sont contenues dans l'intervalle – 32768..32767 (*Smallint*), la variable est stockée sous la forme d'un mot signé.
- Si les deux bornes sont contenues dans l'intervalle 0..65535 (*Word*), la variable est stockée sous la forme d'un mot non signé.
- Si les deux bornes sont contenues dans l'intervalle 2147483648..2147483647 (*Longint*), la variable est stockée sous la forme d'un double mot signé.
- Si les deux bornes sont contenues dans l'intervalle 0..4294967295 (*Longword*), la variable est stockée sous la forme d'un double mot non signé.
- Sinon, la variable est stockée sous la forme d'un quadruple mot signé (*Int64*).

Types caractères

Les variables de type *Char, AnsiChar* ou d'un intervalle du type *Char* sont stockées sous la forme d'un octet non signé. Le type *WideChar* est stocké sous la forme d'un mot non signé.

Types booléens

Un type *Boolean* est stocké sous la forme d'un *Byte*, un type *ByteBool* est stocké sous la forme d'un *Byte*, un type *WordBool* est stocké sous la forme d'un *Word* et un *LongBool* est stocké sous la forme d'un *Longint*.

Un *Boolean* peut prendre les valeurs 0 (*False*) et 1 (*True*). Les types *ByteBool, WordBool* et *LongBool* acceptent les valeurs nulles (*False*) et non nulles (*True*).

Types énumérés

Un type énuméré est stocké sous la forme d'un octet non signé si l'énumération ne comporte pas plus de 256 valeurs et si le type a été déclaré dans l'état {**$Z1**} (ce qui est le cas par défaut). Lorsqu'un type énuméré comporte plus de 256 valeurs ou a été déclaré dans l'état {**$Z2**}, il est stocké sous la forme d'un mot non signé. Si un type énuméré est déclaré dans l'état {**$Z4**}, il est stocké sous la forme d'un mot double non signé.

Types réels

Les types réels stockent la représentation binaire d'un signe, (**+** ou **–**), d'un *exposant* et d'une *mantisse*. Une valeur réelle a la forme :

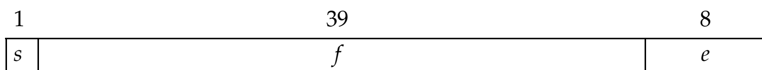
$$+/- \textit{mantisse} * 2^{\textit{exposant}}$$

où la *mantisse* ne possède qu'un bit à gauche de la virgule décimale binaire (c'est-à-dire que $0 <= \textit{mantisse} < 2$).

Dans les schémas suivants, le bit le plus significatif est toujours à gauche et le bit le moins significatif à droite. Les nombres en haut indiquent la largeur (exprimée en bits) de chaque champ, les éléments les plus à gauche étant stockés aux adresses les plus élevées. Par exemple, pour une valeur *Real48*, *e* est stocké dans le premier octet, *f* dans les cinq octets suivants et *s* dans le bit le moins significatif du dernier octet.

Type Real48

Un nombre *Real48* sur 6 octets (48 bits) est décomposé en trois champs :



Si $0 < e <= 255$, la valeur v du nombre est donnée par la formule :

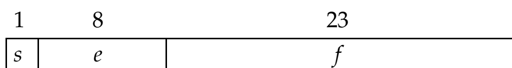
$$v = (-1)^{s} * 2^{(e-129)} * (1.f)$$

Si $e = 0$ alors $v = 0$.

Le type *Real48* ne permet pas de stocker des nombres dénormalisés, NaN ou infinis. Les nombres dénormalisés deviennent nuls quand ils sont stockés dans un *Real48*, et les nombres NaN et infinis produisent une erreur de dépassement de capacité lorsqu'on tente de les stocker dans un *Real48*.

Type Single

Un nombre *Single* sur 4 octets (32 bits) est décomposé en trois champs :



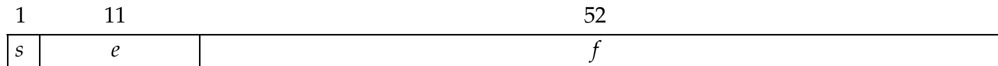
La valeur v du nombre est donnée par la formule :

Si $0 < e < 255$ alors $v = (-1)^s * 2^{(e-127)} * (1.f)$

Si $e = 0$ et $f <> 0$ alors $v = (-1)^s * 2^{(-126)} * (0.f)$
Si $e = 0$ et $f = 0$ alors $v = (-1)^s * 0$
Si $e = 255$ et $f = 0$ alors $v = (-1)^s *$ Inf
Si $e = 255$ et $f <> 0$ alors v est un NaN

Type Double

Un nombre *Double* sur 8 octets (64 bits) est décomposé en trois champs :



La valeur v du nombre est donnée par la formule :

Si $0 < e < 2047$ alors $v = (-1)^s * 2^{(e-1023)} * (1.f)$

Si $e = 0$ et $f <> 0$ alors $v = (-1)^s * 2^{(-1022)} * (0.f)$
Si $e = 0$ et $f = 0$ alors $v = (-1)^s * 0$
Si $e = 2047$ et $f = 0$ alors $v = (-1)^s *$ Inf
Si $e = 2047$ et $f <> 0$ alors v est un NaN

Type Extended

Un nombre *Extended* sur 10 octets (80 bits) est décomposé en quatre champs :

La valeur v du nombre est donnée par la formule :

Si $0 <= e < 32767$ alors $v = (-1)^s * 2^{(e-16383)} * (i.f)$

Si $e = 32767$ et $f = 0$ alors $v = (-1)^s *$ Inf
Si $e = 32767$ et $f <> 0$ alors v est un NaN

Type Comp

Un nombre *Comp* sur 8 octets (64 bits) est stocké comme un entier signé sur 64 bits.

Type Currency

Un nombre *Currency* de 8 octets (64 bits) est stocké sous la forme d'un entier scalaire et signé de 64 bits dont les quatre chiffres les moins significatifs représentent implicitement les quatre chiffres après la virgule.

Types Pointer

Un type *Pointer* est stocké dans 4 octets sous la forme d'une adresse 32 bits. La valeur pointeur **nil** est stockée sous la forme zéro.

Types chaînes courtes

Une chaîne occupe le nombre d'octets correspondant à sa longueur maximum plus un. Le premier octet contient la longueur dynamique réelle de la chaîne et les octets suivants contiennent ses caractères.

L'octet de longueur et les caractères sont des valeurs non signées. La longueur maximum d'une chaîne est de 255 caractères plus un octet de longueur (**string**[255]).

Types chaînes longues

Une variable chaîne longue occupe quatre octets de mémoire, qui contiennent un pointeur sur une chaîne allouée dynamiquement. Lorsqu'une variable chaîne longue est vide (contient une chaîne d'une longueur de zéro), le pointeur de chaîne est **nil** et aucune mémoire dynamique n'est associée avec la variable chaîne. Pour une valeur de chaîne non vide, le pointeur de chaîne pointe sur un bloc de mémoire alloué dynamiquement contenant la valeur chaîne en plus d'un indicateur de longueur 32 bits et d'un compteur de référence 32 bits. Le tableau suivant montre le contenu d'un bloc mémoire de chaîne longue.

Tableau 11.1 Contenu de mémoire dynamique de chaîne longue

Décalage	Contenu
–8	Compteur de référence sur 32 bits
–4	longueur en octets
0..*Length* – 1	Chaîne de caractères
Length	Caractère NULL

Le caractère NULL situé à la fin d'un bloc mémoire de chaîne longue est automatiquement maintenu par le compilateur et par les routines intégrées de gestion de chaîne. Cela permet de transtyper directement une chaîne longue en chaîne à zéro terminal.

Le compilateur génère un bloc mémoire de même implantation qu'une chaîne allouée dynamiquement pour des constantes et littéraux chaînes, mais avec un compteur de référence de -1. Lorsqu'une variable chaîne longue est affectée à une constante chaîne, le pointeur de chaîne se voit affecter l'adresse du bloc mémoire généré pour la constante chaîne. Les routines intégrées de gestion de chaîne savent qu'elles ne doivent pas essayer de modifier des blocs ayant un compteur de référence de –1.

Types chaînes étendues

Remarque Sous Linux, les chaînes étendues sont implémentées exactement comme les chaînes longues.

Sous Windows, une variable chaîne étendue occupe quatre octets de mémoire et contient un pointeur sur une chaîne allouée dynamiquement. Lorsqu'une variable chaîne étendue est vide (contient une chaîne de longueur zéro), le pointeur de chaîne est **nil** et aucune mémoire dynamique n'est associée à la variable chaîne. Pour une valeur de chaîne non vide, le pointeur de chaîne pointe un bloc de mémoire alloué dynamiquement qui contient la valeur de la chaîne en plus d'un indicateur de longueur 32 bits. Le tableau suivant montre le contenu d'un bloc mémoire de chaîne étendue sous Windows.

Tableau 11.2 Contenu de mémoire dynamique de chaîne étendue (Windows seulement)

Décalage	Contenu
–4	Indicateur de longueur, exprimée en octets, sur 32 bits
0..*Length* – 1	Chaîne de caractères
Length	Caractère NULL

La longueur de la chaîne est évaluée en octets, c'est donc le double du nombre de caractères contenus dans la chaîne.

Le caractère NULL à la fin d'un bloc mémoire de chaîne étendue est automatiquement géré par le compilateur et les routines de gestion de chaîne intégrées. Cela rend possible le transtypage direct d'une chaîne étendue par une chaîne à zéro terminal.

Types ensembles

Un ensemble est un tableau de bits dans lequel chaque bit indique si un élément est présent dans l'ensemble ou non. Le nombre maximum d'éléments dans un ensemble est 256, et un ensemble n'occupe donc jamais plus de 32 octets. Le nombre d'octets occupés par un ensemble particulier est calculé comme suit :

(*Max* div 8) – (*Min* div 8) + 1

où *Min* et *Max* sont les bornes inférieure et supérieure du type de base de cet ensemble. Le numéro de l'octet d'un élément particulier *E* est :

(*E* div 8) – (*Min* div 8)

Et le numéro du bit à l'intérieur de l'octet est :

E mod 8

où *E* désigne la valeur scalaire de l'élément. Quand c'est possible, le compilateur stocke les ensembles dans les registres CPU, mais un ensemble réside toujours en mémoire s'il est plus grand que le type *Integer* générique ou si le programme contient du code qui utilise l'adresse de l'ensemble.

Types tableaux statiques

Un tableau statique est stocké sous la forme d'une suite contiguë de variables du type des éléments du tableau. Les éléments d'indices les plus faibles sont stockés aux adresses mémoire les plus faibles. Dans un tableau multidimensionnel, la dimension la plus à droite est celle qui s'incrémente en premier.

Types tableaux dynamiques

Une variable tableau dynamique occupe quatre octets de mémoire qui contiennent un pointeur sur le tableau alloué dynamiquement. Quand la variable est vide (non initialisée) ou contient un tableau de longueur nulle, le pointeur vaut **nil** et il n'y a pas de mémoire dynamique associée à la variable. Pour un tableau non vide, la variable pointe sur un bloc de mémoire alloué dynamiquement qui contient en plus du tableau un indicateur de longueur sur 32 bits et un compteur de référence sur 32 bits. Le tableau suivant indique l'organisation du bloc de mémoire de tableau alloué dynamiquement.

Tableau 11.3 Organisation de la mémoire d'un tableau dynamique

Décalage	Contenu
–8	Compteur de référence sur 32 bits
–4	Indicateur de longueur sur 32 bits (nombre d'éléments)
0..*Length* * (taille d'élément) – 1	Eléments du tableau

Types enregistrements

Lorsqu'un type d'enregistrements est déclaré dans l'état **{$A+}** (valeur par défaut) et lorsque la déclaration ne comprend pas de modificateur **packed**, le type est un *enregistrement décompacté*, et les champs de l'enregistrement sont alignés pour faciliter l'accès par la CPU. L'alignement est contrôlé par le type de chaque champ. Chaque type de données a un alignement inhérent, automatiquement calculé par le compilateur. L'alignement peut être 1, 2, 4 ou 8 et représente la limite en octets sur laquelle une valeur du type doit être stockée pour que l'accès soit plus efficace. Le tableau suivant liste les alignements pour tous les types de données.

Tableau 11.4 Masques d'alignement des types

Type	Alignement
Types scalaires	Taille du type (1, 2, 4 ou 8)
Types réels	2 pour *Real48*, 4 pour *Single*, 8 pour *Double* et *Extended*
Types chaîne courte	1
Types tableau	Identique au type des éléments du tableau
Types enregistrement	Le plus grand alignement des champs de l'enregistrement

Tableau 11.4 Masques d'alignement des types

Type	Alignement
Types ensemble	Taille du type si 1, 2 ou 4, sinon 1
Tous les autres types	4

Pour que l'alignement des champs soit correct dans un type enregistrement décompacté, le compilateur insère un octet inutilisé avant les champs ayant un alignement de 2, et jusqu'à trois octets inutilisés avant les champs ayant un alignement de 4, si nécessaire. Enfin, le compilateur arrondit la taille totale de l'enregistrement jusqu'à la limite en octets spécifiée par l'alignement le plus grand des champs.

Lorsqu'un type enregistrement est déclaré dans l'état **{$A–}** ou lorsque la déclaration comprend le modificateur **packed**, les champs de l'enregistrement ne sont pas alignés, mais des déplacements consécutifs leur sont plutôt affectés. La taille totale d'un tel enregistrement compacté est simplement la taille de tous les champs. Comme l'alignement des données peut changer, c'est une bonne idée de compacter toute structure enregistrement que vous avez l'intention d'écrire sur disque ou de passer en mémoire à un autre module compilé avec une version du compilateur différente.

Types fichiers

Les types de fichiers sont représentés sous forme d'enregistrements. Les fichiers typés et non typés occupent 332 octets, répartis comme suit :

```
type
  TFileRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
    case Byte of
      0: (RecSize: Cardinal);
      1: (BufSize: Cardinal;
          BufPos: Cardinal;
          BufEnd: Cardinal;
          BufPtr: PChar;
          OpenFunc: Pointer;
          InOutFunc: Pointer;
          FlushFunc: Pointer;
          CloseFunc: Pointer;
          UserData: array[1..32] of Byte;
          Name: array[0..259] of Char; );
  end;
```

Les fichiers texte occupent 460 octets, répartis comme suit :

```
type
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle: Integer;
    Mode: word;
```

```
    Flags: word;
    BufSize: Cardinal;
    BufPos: Cardinal;
    BufEnd: Cardinal;
    BufPtr: PChar;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of Char;
    Buffer: TTextBuf;
  end;
```

Handle contient le handle du fichier (quand celui-ci est ouvert).

Le champ *Mode* peut prendre l'une des valeurs suivantes :

```
const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

où *fmClosed* indique que le fichier est fermé. *fmInput* et *fmOutput* indiquent un fichier texte qui a été réinitialisé (*fmInput*) ou réécrit (*fmOutput*), *fmInOut* indique un fichier typé ou non typé qui a été réinitialisé ou réécrit. Toute autre valeur indique que la variable fichier n'a pas été affectée (et qu'elle n'est donc pas initialisée).

Le champ *UserData* est disponible pour les routines écrites par l'utilisateur afin d'y stocker des données.

Name contient le nom du fichier ; il s'agit d'une suite de caractères terminée par un caractère null (#0).

Pour les fichiers typés et non typés, *RecSize* contient la longueur de l'enregistrement en octets, et le champ *Private* est inutilisé mais réservé.

Pour les fichiers texte, *BufPtr* est un pointeur sur un tampon de *BufSize* octets, *BufPos* est l'indice dans le tampon du prochain caractère à lire à ou écrire, et *BufEnd* représente le nombre de caractères valides dans le tampon. *OpenFunc*, *InOutFunc*, *FlushFunc* et *CloseFunc* sont des pointeurs sur les routines d'E/S qui contrôlent le fichier. Pour davantage d'informations, voir “Fonctions de périphérique” à la page 8-6. Les indicateurs déterminent le style de rupture de ligne comme suit :

bit 0 vide	rupture de ligne LF
bit 0 défini	rupture de ligne CRLF

Tous les autres bits indicateurs sont réservés pour un usage ultérieur. Voir aussi *DefaultTextLineBreakStyle* et *SetLineBreakStyle*.

Types procédures

Un pointeur de procédure est stocké sous la forme d'un pointeur 32 bits sur le point d'entrée d'une procédure ou d'une fonction. Un pointeur de méthode est stocké sous la forme d'un pointeur 32 bits sur le point d'entrée d'une méthode, suivi par un pointeur 32 bits sur un objet.

Types classes

Une valeur d'un type classe est stockée sous la forme d'un pointeur 32 bits sur une instance de la classe (ce qu'on appelle aussi un *objet*). Le format interne d'un objet est similaire à celui d'un enregistrement. Les champs d'un objet sont stockés sous la forme d'une suite contiguë de variables, dans l'ordre de leur déclaration. Les champs sont toujours alignés, correspondant à un type enregistrement décompacté. Les champs hérités d'une classe ancêtre sont stockés avant les nouveaux champs définis dans la classe descendante.

Les quatre premiers octets de chaque objet contiennent un pointeur sur la *table des méthodes virtuelles* (VMT) de la classe. Il n'existe qu'une seule VMT par classe (et non une par objet), mais deux types de classes distincts ne partagent jamais une VMT, quel que soit leur degré de similitude. Les VMT sont construites automatiquement par le compilateur et ne sont jamais manipulées directement par un programme. De même, les pointeurs sur les VMT qui sont stockés automatiquement dans les objets par la méthode constructeur ne sont jamais directement manipulés par un programme.

Le contenu d'une VMT est présenté dans le tableau ci-dessous. Aux décalages positifs, une VMT se compose d'une liste de pointeurs de méthodes sur 32 bits, un par méthode virtuelle définie par l'utilisateur dans le type classe, dans l'ordre de la déclaration. Chaque emplacement contient l'adresse du point d'entrée de la méthode virtuelle correspondante. Cette structure est compatible avec une v-table C++ ainsi qu'avec COM. Aux décalages négatifs, une VMT contient plusieurs champs internes à Pascal Objet. Les applications doivent utiliser les méthodes définies dans *TObject* pour obtenir ces informations, car cette structure pourra être amenée à changer dans les versions futures de Pascal Objet.

Tableau 11.5 Organisation de la table des méthodes virtuelles

Offset	Type	Description
–76	*Pointer*	Pointeur sur la table des méthodes virtuelles (ou **nil**)
–72	*Pointer*	Pointeur sur la table d'interface (ou **nil**)
–68	*Pointer*	Pointeur sur la table des informations d'automation (ou **nil**)
–64	*Pointer*	Pointeur sur la table d'initialisation d'instance (ou **nil**)
–60	*Pointer*	Pointeur sur la table des informations de type (ou **nil**)
–56	*Pointer*	Pointeur sur la table de définition des champs (ou **nil**)
–52	*Pointer*	Pointeur sur la table de définition des méthodes (ou **nil**)
–48	*Pointer*	Pointeur sur la table des méthodes dynamiques (ou **nil**)
–44	*Pointer*	Pointeur sur une chaîne courte contenant le nom de la classe

Tableau 11.5 Organisation de la table des méthodes virtuelles

Offset	Type	Description
–40	*Cardinal*	Taille de l'instance en octets
–36	*Pointer*	Pointeur sur un pointeur vers la classe ancêtre (ou **nil**)
–32	*Pointer*	Pointeur sur le point d'entrée de la méthode *SafecallException* (ou **nil**)
–28	*Pointer*	Point d'entrée de la méthode *AfterConstruction*
–24	*Pointer*	Point d'entrée de la méthode *BeforeDestruction*
–20	*Pointer*	Point d'entrée de la méthode *Dispatch*
–16	*Pointer*	Point d'entrée de la méthode *DefaultHandler*
–12	*Pointer*	Point d'entrée de la méthode *NewInstance*
–8	*Pointer*	Point d'entrée de la méthode *FreeInstance*
–4	*Pointer*	Point d'entrée du destructeur *Destroy*
0	*Pointer*	Point d'entrée de la première méthode virtuelle définie par l'utilisateur
4	*Pointer*	Point d'entrée de la seconde méthode virtuelle définie par l'utilisateur
⋮	⋮	⋮

Types références de classe

Une valeur référence de classe est stockée sous la forme d'un pointeur 32 bits sur la table des méthodes virtuelles (VMT) d'une classe.

Types variants

Un variant est stocké sous la forme d'un enregistrement de 16 octets contenant un code de type et une valeur de type (ou une référence à une valeur), selon le type donné par le code type. Les unités *System* et *Variants* définissent des constantes et des types pour les variants.

Le type *TVarData* représente la structure interne d'une variable *Variant* (sous Windows, il est identique au type *Variant* utilisé par COM et l'API Win32). Le type *TVarData* peut être utilisé dans des transtypages de variables *Variant* pour accéder à la structure interne d'une variable.

Le champ VType d'un enregistrement *TVarData* contient le code type du variant dans les douze bits inférieurs (bits définis par la constante *varTypeMask*). De plus, le bit *varArray* peut être défini pour indiquer que le variant est un tableau, et le bit *varByRef* pour indiquer que le variant contient une référence (par opposition à une valeur).

Les champs *Reserved1*, *Reserved2* et *Reserved3* d'un enregistrement *TVarData* sont inutilisés.

Le contenu des huit octets restants d'un enregistrement *TVarData* dépend du champ *VType*. Si les bits *varArray* et *varByRef* ne sont pas définis, le variant contient une valeur du type donné.

Si le bit *varArray* est défini, le variant contient un pointeur sur une structure *TVarArray* définissant le tableau. Le type de chaque élément tableau est donné par les bits *varTypeMask* dans le champ *VType*.

Si le bit *varByRef* est défini, le variant contient une référence à une valeur du type donné par les bits *varTypeMask* et *varArray* dans le champ *VType*.

Notez que le code type *varString* est privé. Les variants contenant une valeur *varString* ne doivent jamais être passés à une fonction non Delphi. Sous Windows, le support Automation de Delphi convertit automatiquement les variants *varString* en variants *varOleStr* avant de les transmettre sous forme de paramètres aux fonctions externes.

Chapitre 12

Contrôle des programmes

Ce chapitre explique comment les paramètres et le résultat des fonctions sont stockés et transmis. La dernière section traite des procédures de sortie.

Paramètres et résultats de fonction

Le traitement des paramètres et du résultat des fonctions est déterminé par plusieurs facteurs dont la convention d'appel, la sémantique des paramètres et le type ou la taille de la valeur transmise.

Transfert de paramètre

Les paramètres sont transmis aux procédures et aux fonctions par l'intermédiaire des registres CPU ou de la pile, selon la convention d'appel de la routine. Pour davantage d'informations sur les conventions d'appel, voir "Conventions d'appel" à la page 6-5.

Les paramètres variable (**var**) sont toujours transmis par référence. Il s'agit de pointeurs 32 bits indiquant l'emplacement de stockage réel.

Les paramètres valeur et constante (**const**) sont transmis par valeur ou par référence, selon le type et la taille du paramètre :

- Un paramètre scalaire est transmis sous la forme de valeur 8 bits, 16 bits, 32 bits ou 64 bits, en utilisant le même format qu'une variable de type correspondant.
- Un paramètre réel est toujours transmis sur la pile. Un paramètre *Single* occupe 4 octets, et un paramètre *Double*, *Comp* ou *Currency* occupe 8 octets. Un paramètre *Real48* occupe 8 octets, avec la valeur *Real48* stockée dans les 6 octets inférieurs. Un paramètre *Extended* occupe 12 octets, avec la valeur *Extended* stockée dans les 10 octets inférieurs.

- Un paramètre chaîne courte est transmis sous la forme d'un pointeur 32 bits sur une chaîne courte.
- Un paramètre de chaîne longue ou tableau dynamique est transmis sous la forme d'un pointeur 32 bits sur le bloc mémoire dynamique alloué pour la chaîne longue. La valeur **nil** est transmise pour une chaîne longue vide.
- Un paramètre pointeur, classe, référence de classe ou pointeur de procédure global est transmis sous la forme d'un pointeur 32 bits
- Un pointeur de méthode est transmis sur la pile sous la forme de pointeurs 32 bits. Le pointeur d'instance est empilé avant le pointeur de méthode afin que ce dernier occupe la plus petite adresse.
- A*vec les conventions* ***register*** *et* ***pascal****, u*n paramètre variant est transmis sous la forme d'un pointeur 32 bits à une valeur *Variant*.
- Les tableaux, les enregistrements et les ensembles de 1, 2 ou 4 octets sont transmis sous la forme de valeurs 8 bits, 16 bits et 32 bits. Les tableaux, enregistrements et ensembles plus grands sont transmis sous la forme de pointeurs 32 bits sur la valeur. Exception à cette règle : les enregistrements sont toujours transmis directement sur la pile avec les conventions **cdecl**, **stdcall et safecall**. La taille d'un enregistrement transmis de cette manière est toujours arrondie à la limite supérieure la plus proche de deux mots.
- Un paramètre tableau ouvert est transmis sous la forme de valeurs 32 bits. La première valeur 32 bits est un pointeur sur les données du tableau. La deuxième valeur 32 bits représente le nombre d'éléments du tableau moins un.

Quand deux paramètres sont transmis dans la pile, chacun d'entre eux occupe un multiple de 4 octets (un nombre entier de deux mots). Pour un paramètre 8 bits ou 16 bits, même si le paramètre n'occupe qu'un octet ou un mot, il est transmis sous la forme d'un double mot. Le contenu des parties inutilisées du double mot n'est pas défini.

Avec les conventions **pascal**, **cdecl**, **stdcall et safecall**, tous les paramètres sont transmis sur la pile. Pour la convention **pascal**, les paramètres sont empilés dans leur ordre de déclaration (gauche vers droite), afin que le premier paramètre se termine à l'adresse la plus haute et que le dernier paramètre se termine à l'adresse la plus basse. Pour les conventions **cdecl, stdcall** et **safecall**, les paramètres sont empilés dans l'ordre inverse de leur déclaration (droite vers gauche), afin que le premier paramètre se termine à l'adresse la plus basse et que le dernier paramètre se termine à l'adresse la plus haute.

Avec la convention **register**, jusqu'à trois paramètres sont transmis dans les registres CPU et le reste (s'il existe) est transmis sur la pile. Les paramètres sont transmis dans l'ordre de leur déclaration (comme avec la convention **pascal**). Les trois premiers paramètres retenus sont transmis respectivement dans les registres EAX, EDX et ECX. Les types réels, pointeur de méthode, variant, *Int64* et structurés ne sont pas retenus comme paramètres registre, mais tous les autres types de paramètre le sont. Si plus de trois paramètres sont utilisables comme paramètres registre, les trois premiers sont transmis dans EAX, EDX et ECX, et les paramètres restants sont empilés sur la pile dans l'ordre de leur déclaration.

Par exemple, soit la déclaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

un appel à *Test* transmet *A* dans EAX sous la forme d'un entier 32 bits, *B* dans EDX sous la forme d'un pointeur sur un *Char*, et *D* dans ECX sous la forme d'un pointeur sur un bloc mémoire de chaîne longue. De plus, *C* et *E* sont empilés, dans cet ordre, sous la forme, respectivement d'un double mot et d'un pointeur 32 bits.

Règles d'enregistrement des registres

Les procédures et les fonctions doivent préserver la valeur des registres EBX, ESI, EDI et EBP, mais, elles peuvent modifier les registres EAX, EDX et ECX. Quand vous implémentez un constructeur ou un destructeur en assembleur, assurez-vous de préserver le registre DL. Les procédures et les fonctions sont toujours appelées en supposant que l'indicateur de direction de la CPU est effacé (ce qui correspond à une instruction CLD), et, à la sortie des routines, cet indicateur doit également être effacé.

Résultats de fonction

Les conventions suivantes sont utilisées pour renvoyer des valeurs de résultat de fonction :

- Les résultats scalaires sont renvoyés, quand c'est possible, dans un registre CPU : les octets sont renvoyés dans AL, les mots dans AX, les doubles mots dans EAX.
- Les résultats réels sont renvoyés dans le registre du haut de la pile du coprocesseur à virgule flottante (ST(0)). Pour les résultats de fonction de type *Currency*, la valeur dans ST(0) est graduée par 10 000. Par exemple, la valeur *Currency* 1.234 est renvoyée en ST(0) en 12340.
- Pour un résultat chaîne, pointeur de méthode, variant ou *Int64,* les effets sont les mêmes que si le résultat de la fonction était déclaré en tant que paramètre **var** supplémentaire suivant les paramètres déclarés. En d'autres termes, l'appelant passe un pointeur 32 bits supplémentaire qui pointe sur une variable dans laquelle le résultat de la fonction doit être renvoyé.
- Les résultats pointeur, classe, référence de classe et pointeur de procédure global sont renvoyés dans EAX.
- Pour les résultats tableau statique, enregistrement et ensemble, si la valeur occupe un octet, elle est retournée dans AL. Si la valeur occupe deux octets, elle est renvoyée dans AX, et si la valeur occupe quatre octets, elle est renvoyée dans EAX. Sinon le résultat est renvoyé dans un paramètre **var** supplémentaire et transmis à la fonction après les paramètres déclarés.

Appels de méthode

Les méthodes emploient les mêmes conventions d'appel que les procédures et les fonctions ordinaires. Cependant, elles disposent d'un paramètre implicite supplémentaire nommé *Self*, qui correspond à une référence sur la classe ou l'instance dans laquelle la méthode a été appelée. Le paramètre *Self* est transmis sous la forme d'un pointeur de 32 bits.

- Avec la convention **register**, *Self* se comporte comme s'il avait été déclaré *avant tous les autres paramètres*. Il est par conséquent toujours transmis dans le registre EAX.
- Avec la convention **pascal**, *Self* se comporte comme s'il avait été déclaré *après* tous les autres paramètres (y compris le paramètre supplémentaire **var** qui peut être transmis pour obtenir un résultat de fonction). Il est donc toujours empilé en dernier, se terminant à une adresse inférieure à celle de tous les autres paramètres.
- Pour les conventions **cdecl, stdcall et safecall**, *Self* se comporte comme s'il avait été déclaré *avant* tous les autres paramètres, mais *après* le paramètre supplémentaire **var** qui doit être transmis pour obtenir un résultat de fonction. Il est donc toujours empilé en dernier, mais avant le paramètre supplémentaire **var**.

Constructeurs et destructeurs

Les constructeurs et les destructeurs utilisent les mêmes conventions d'appel que les autres méthodes, si ce n'est qu'un paramètre indicateur *Boolean* est transmis pour indiquer le contexte de l'appel constructeur ou destructeur.

Une valeur *False* du paramètre indicateur dans un appel constructeur indique que ce dernier a été appelé via une instance d'objet ou avec le mot-clé **inherited**. Dans ce cas, le comportement du constructeur est celui d'une méthode ordinaire.

Une valeur *True* du paramètre indicateur dans un appel constructeur indique que ce dernier a été appelé via une référence de classe. Dans ce cas, le constructeur crée une instance de la classe fournie par *Self*, et renvoie une référence sur l'objet nouvellement créé dans EAX.

Une valeur *False* du paramètre indicateur dans un appel destructeur indique que ce dernier a été appelé en utilisant le mot-clé **inherited**. Dans ce cas, le comportement du destructeur est celui d'une méthode ordinaire.

Une valeur *True* dans le paramètre indicateur dans un appel destructeur indique que ce dernier a été appelé via une instance d'objet. Dans ce cas, le destructeur désalloue l'instance donnée par *Self* avant de rendre la main.

Le paramètre indicateur se comporte comme s'il avait été déclaré avant tous les autres paramètres. Avec la convention **register**, l'indicateur est toujours transmis dans le registre DL, avec la convention **pascal**, l'indicateur est toujours empilé avant tous les autres paramètres, et avec les conventions **cdecl**, **stdcall et safecall**, l'indicateur est toujours empilé juste devant le paramètre *Self*.

Comme le registre DL indique si le constructeur ou le destructeur est à l'extrémité de la pile d'appels, vous devez restaurer la valeur de DL avant de sortir pour que *BeforeDestruction* ou *AfterConstruction* puissent être appelés correctement.

Procédures de sortie

Les procédures de sortie garantissent que des actions spécifiques sont effectuées (par exemple l'enregistrement et la fermeture de fichiers) avant l'arrêt d'un programme. La variable pointeur *ExitProc* vous permet “d'installer” une procédure de sortie afin qu'elle soit systématiquement appelée lors de l'arrêt du programme et ce, que l'arrêt soit normal, forcé par l'appel de *Halt* ou le résultat d'une erreur d'exécution. Une procédure de sortie n'attend aucun paramètre.

Remarque Pour définir des comportements de sortie, il est conseillé d'utiliser les sections finalisation plutôt que les procédures de sortie, voir “Section finalisation” à la page 3-5. Les procédures de sortie ne sont utilisables que pour les cibles exécutables, objets partagés (Linux) ou DLL (Windows) ; pour les paquets, le comportement de sortie doit être implémenté dans la section finalisation. Toutes les procédures de sortie sont appelées avant les sections finalisation.

Les unités peuvent, tout comme les programmes, installer des procédures de sortie. Une unité peut installer une procédure de sortie dans son code d'initialisation et se reposer sur la procédure pour fermer les fichiers ou accomplir d'autres opérations de remise en ordre.

Quand elle est correctement implémentée, une procédure de sortie fait partie d'une chaîne de procédures de sortie. Les procédures sont exécutées dans l'ordre inverse de celui de leur installation, ce qui garantit que le code de sortie d'une unité n'est pas exécuté avant celui des unités qui dépendent d'elle. Pour conserver la chaîne intacte, vous devez enregistrer la valeur en cours de *ExitProc* avant de lui affecter l'adresse de votre propre procédure de sortie. Ainsi, la première instruction de votre procédure de sortie peut réinstaller la valeur enregistrée de *ExitProc*.

Le code suivant illustre le squelette de l'implémentation d'une procédure de sortie :

```
var
  ExitSave: Pointer;

procedure MaSortie;
begin
  ExitProc := ExitSave;  // toujours commencer par rétablir l'ancien vecteur
  ⋮
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MaSortie;
  ⋮
end.
```

A l'arrivée, le code enregistre le contenu de *ExitProc* dans *ExitSave,* puis installe la procédure *MaSortie*. Quand elle est appelée lors de l'arrêt du programme, *MaSortie commence par* réinstaller la précédente procédure de sortie.

La routine de terminaison de programme de la bibliothèque d'exécution continue à appeler les procédures de sortie jusqu'à ce que *ExitProc* prenne la valeur **nil**. Afin d'éviter des boucles infinies, *ExitProc* est initialisée à **nil** avant chaque appel, de manière à ce que la procédure de sortie suivante ne soit appelée que si la procédure de sortie en cours affecte réellement une adresse à *ExitProc*. Si une erreur survient pendant l'exécution d'une procédure de sortie, celle-ci n'est plus appelée de nouveau.

Pour connaître la cause de la terminaison du programme, la procédure de sortie peut lire la variable entière *ExitCode* et la variable pointeur *ErrorAddr*. En cas de terminaison normale, *ExitCode* vaut zéro et *ErrorAddr* vaut **nil**. Dans le cas d'un arrêt provoqué par un appel à *Halt*, *ExitCode* contient la valeur transmise à *Halt* et *ErrorAddr* vaut **nil**. Enfin, si l'arrêt du programme est consécutif à une erreur d'exécution, *ExitCode* reçoit le code d'erreur et *ErrorAddr* l'adresse de l'instruction incorrecte.

La dernière procédure de sortie (celle installée par la bibliothèque d'exécution) ferme les fichiers *Input* et *Output* et génère un message d'erreur si *ErrorAddr* est différent de **nil**. Pour afficher vous-même les messages d'erreur d'exécution, installez une procédure de sortie qui lit le contenu de *ErrorAddr* et affiche un message s'il n'est pas à **nil**. De plus, avant la sortie de cette procédure, affectez la valeur **nil** à *ErrorAddr* afin que l'erreur éventuelle ne soit pas à nouveau signalée par d'autres procédures de sortie.

Après avoir appelé toutes les procédures de sortie, la bibliothèque d'exécution rend la main au système d'exploitation et renvoie la valeur contenue dans *ExitCode*.

Chapitre 13

Code assembleur en ligne

L'assembleur intégré vous permet d'écrire du code assembleur Intel dans des programmes Pascal Objet. Il implémente un large sous-ensemble de la syntaxe reconnue par Turbo Assembleur et le Macro Assembleur Microsoft, y compris tous les codes opératoires 8086/8087 et 80386/80387 et presque tous les opérateurs d'expression du Turbo Assembleur. De plus, l'assembleur intégré vous permet d'utiliser des identificateurs Pascal Objet dans des instructions en assembleur.

A l'exception de DB, DW et de DD (définition d'octet, de mot et de double mot), aucune des directives du Turbo Assembleur, telles que EQU, PROC, STRUC, SEGMENT et MACRO, n'est supportée par l'assembleur intégré. Les opérations implémentées via des directives Turbo Assembleur ont une construction Pascal Objet correspondante. Par exemple, la plupart des directives EQU correspondent aux déclarations de constante, de variable et de type, la directive PROC correspond aux déclarations de procédure et de fonction, et la directive STRUC correspond au type enregistrement (**record**).

Au lieu d'utiliser l'assembleur intégré, vous pouvez lier des fichiers objet contenant des procédures et fonctions externes. Pour davantage d'informations, voir “Liaison de fichiers objet” à la page 6-7.

Remarque Si vous voulez utiliser du code assembleur externe dans vos applications, vous devez envisager de le réécrire en Pascal Objet ou au moins de le réimplémenter en utilisant l'assembleur inline.

L'instruction asm

L'accès à l'assembleur intégré se fait par l'intermédiaire de l'instruction **asm** qui a la forme suivante :

`asm` *listeInstruction* `end`

où *listeInstruction* est une suite d'instructions assembleur séparées par des points-virgules, des caractères de fin de ligne ou des commentaires Pascal Objet.

Les commentaires placés à l'intérieur d'une instruction **asm** doivent se faire en utilisant le style Pascal Objet. Un point-virgule n'indique pas que le reste de la ligne est un commentaire.

Le mot réservé **inline** et la directive **assembler** ne sont conservés que dans un souci de compatibilité ascendante. Ils sont ignorés par le compilateur.

Utilisation des registres

En général, les règles d'utilisation de registre dans une instruction **asm** sont les mêmes que celles d'une procédure ou d'une fonction **external**. Une instruction **asm** doit conserver les registres EDI, ESI, ESP, EBP et EBX, mais peut librement modifier les registres EAX, ECX et EDX. A l'entrée d'une instruction **asm**, BP pointe sur le cadre de pile actuel, SP pointe sur le haut de la pile, SS contient l'adresse de segment du segment de pile et DS contient l'adresse de segment du segment de données. A l'exception de EDI, ESI, ESP, EBP et EBX, une instruction **asm** peut présumer qu'il n'y a rien dans le contenu du registre à l'entrée de l'instruction.

Syntaxe des instructions assembleur

Une instruction assembleur a la syntaxe suivante :

Label: *Préfixe* *CodeOpératoire* $Opérande_1$, $Opérande_2$

où *Label* est un label, *Préfixe* est un code opératoire préfixe assembleur, *CodeOpératoire* est un code opératoire d'instruction assembleur ou une directive et *Opérande* est une expression assembleur. *Label* et *Préfixe* sont facultatifs. Certains codes opératoires ne prennent qu'un seul opérande, d'autres n'en prennent aucun.

Les commentaires sont autorisés entre des instructions assembleur, mais pas à l'intérieur d'elles. Par exemple :

```
MOV AX,1 {Valeur initiale}     { OK }
MOV CX,100 {Compteur}          { OK }

MOV {Valeur initiale} AX,1;    { Erreur! }
MOV CX, {Compteur} 100         { Erreur! }
```

Labels

Les labels sont utilisés dans les instructions de l'assembleur intégré comme ils le sont en Pascal Objet : en écrivant un label suivi du caractère deux-points avant une instruction. Il n'y a pas de limite à la longueur d'un label mais seul les 32 premiers caractères sont significatifs. Comme en Pascal Objet, les labels doivent être déclarés dans la partie déclaration **label** du bloc contenant l'instruction **asm**. Il y a une exception à cette règle : *les labels locaux*.

Les labels locaux sont des labels qui commencent par le signe @. Ils sont composés du signe @ suivi d'une ou de plusieurs lettres, chiffres, caractères souligné ou du signe @. L'utilisation des labels locaux est limitée aux instructions **asm** et la portée d'un label local commence avec le mot réservé **asm** jusqu'à la fin de l'instruction **asm** qui le contient. Il n'est pas nécessaire de déclarer un label local.

Codes opératoires d'instruction

L'assembleur intégré supporte tous les codes opératoires documentés par Intel pour l'utilisation générale dans les applications. Notez que des instructions privilégiées du système d'exploitation peuvent ne pas être supportées. Les familles d'instructions suivantes sont supportées :

- famille Pentium
- Pentium Pro
- MMX
- SIMD
- SSE
- AMD 3DNow!

Pour une description complète de chaque instruction, consultez la documentation de votre microprocesseur.

Dimension de l'instruction RET

Le code opératoire d'instruction RET génère toujours un renvoi proche.

Dimension du saut automatique

A moins qu'il ne soit dirigé autrement, l'assembleur intégré optimise les instructions de saut en sélectionnant automatiquement la forme la plus courte, et par conséquent la plus efficace d'une instruction de saut. Cette dimension de saut automatique s'applique à l'instruction de saut inconditionnel (JMP), et à toutes les instructions de saut conditionnel quand la cible est une étiquette (et non une procédure ou une fonction).

Pour une instruction de saut inconditionnel (JMP), l'assembleur intégré génère un saut court (un code opératoire d'un octet suivi d'un octet de déplacement) si la distance pour l'étiquette cible est comprise entre –128 et 127 octets. Sinon l'assembleur intégré génère un saut de proximité (un code opératoire d'un octet suivi d'un déplacement de deux octets) est généré.

Pour une instruction de saut conditionnel, un saut court (un code opératoire d'un octet suivi d'un octet de déplacement) est généré si la distance à l'étiquette cible est comprise entre –128 et 127 octets. Sinon, l'assembleur intégré génère un saut court avec la condition inverse, qui passe un saut near jusqu'à l'étiquette cible (cinq octets au total). Par exemple, l'instruction assembleur suivante :

```
JC      Stop
```

où *Stop* n'est pas à la portée d'un saut court et est converti en séquence de code machine correspondant à :

```
JNC      Skip
JMP      Stop
Skip:
```

Les sauts sur les points d'entrée de procédures et fonctions sont toujours proches.

Directives assembleur

L'assembleur intégré de Delphi supporte trois directives assembleur : DB (définition d'octet), DW (définition de mot) et DD (définition de double mot). Elles génèrent chacune les données correspondant aux opérandes séparées par des virgules qui suivent la directive.

La directive DB génère une séquence d'octets. Chaque opérande peut être une expression constante avec une valeur comprise entre –128 et 255 ou une chaîne de caractères de n'importe quelle longueur. Les expressions constantes génèrent un octet de code et les chaînes génèrent une séquence d'octets avec des valeurs correspondant au code ASCII de chaque caractère.

La directive DW génère une séquence de mots. Chaque opérande peut être une expression constante avec une valeur comprise entre –32 768 et 65 535 ou une expression adresse. Pour une expression adresse, l'assembleur intégré génère un pointeur near, c'est-à-dire un mot contenant la partie déplacement de l'adresse.

La directive DD génère une séquence de double mots. Chaque opérande peut être une expression constante avec une valeur comprise entre –2 147 483 648 et 4 294 967 295 ou une expression adresse. Pour une expression adresse, l'assembleur intégré génère un pointeur far, c'est-à-dire, un mot contenant la partie déplacement de l'adresse, suivi d'un mot contenant la partie segment de l'adresse.

La directive DQ définit un mot quadruple pour les valeurs Int64.

Les données générées par les directives DB, DW et DD sont toujours stockées dans le segment de code, tout comme le code généré par d'autres instructions de l'assembleur intégré. Pour générer des données non initialisées ou initialisées dans le segment de données, vous devez utiliser les déclarations Pascal Objet **var** ou **const**.

Voici des exemples de directives DB, DW et DD :

```
asm
  DB     0FFH                      { Un octet}
  DB     0,99                      { Deux octets}
  DB     'A'                       { Ord('A') }
  DB     'Hello world...',0DH,0AH  { Chaîne suivie par CR/LF }
  DB     12,"chaîne"               { Chaîne de style Pascal Objet }
  DW     0FFFFH                    { Un mot}
  DW     0,9999                    { Deux mots}
  DW     'A'                       { Idem DB 'A',0 }
  DW     'BA'                      { Idem DB 'A','B' }
  DW     MyVar                     { Déplacement de MyVar }
  DW     MyProc                    { Déplacement de MyProc }
  DD     0FFFFFFFFH                { Un double mot}
  DD     0,999999999               { Deux double mots}
  DD     'A'                       { Idem DB 'A',0,0,0 }
  DD     'DCBA'                    { Idem DB 'A','B','C','D' }
  DD     MyVar                     { Pointeur sur MyVar }
  DD     MyProc                    { Pointeur sur MyProc }
end;
```

En Turbo Assembleur, lorsqu’un identificateur précède une directive DB, DW ou DD, il provoque la déclaration d’une variable octet, mot ou double mot à l’emplacement de la directive. Par exemple, Turbo Assembleur autorise :

```
ByteVar     DB     ?
WordVar     DW     ?
IntVar      DD     ?
  ⋮
            MOV    AL,ByteVar
            MOV    BX,WordVar
            MOV    ECX,IntVar
```

L’assembleur intégré ne supporte pas de telles déclarations de variable. Le seul type de symbole pouvant être défini dans une instruction de l’assembleur intégré est une étiquette. Toutes les variables doivent être déclarées en utilisant la syntaxe Pascal Objet, et la construction précédente peut être remplacée par :

```
var
  ByteVar: Byte;
  WordVar: Word;
  IntVar: Integer;
  ⋮
asm
  MOV    AL,ByteVar
  MOV    BX,WordVar
  MOV    ECX,IntVar
end;
```

Opérandes

Les opérandes de l'assembleur intégré sont des expressions composées de constantes, de registres, de symboles et d'opérateurs.

Parmi les opérandes, les mots réservés suivants ont une signification particulière :

Tableau 13.1 Mots réservés de l'assembleur intégré

AH	BX	DI	EBX	ESP	OFFSET	SP
AL	BYTE	DL	ECX	FS	OR	SS
AND	CH	DS	EDI	GS	PTR	ST
AX	CL	DWORD	EDX	HIGH	QWORD	TBYTE
BH	CS	DX	EIP	LOW	SHL	TYPE
BL	CX	EAX	ES	MOD	SHR	WORD
BP	DH	EBP	ESI	NOT	SI	XOR

Les mots réservés ont toujours la priorité sur les identificateurs définis par l'utilisateur. Ainsi :

```
var
  Ch: Char;
  ⋮
asm
  MOV    CH, 1
end;
```

charge 1 dans le registre CH, et *non* dans la variable *Ch*. Pour accéder à un symbole défini par l'utilisateur ayant le même nom qu'un mot réservé, vous devez utiliser l'opérateur de surcharge d'identificateur **&** :

```
MOV    &Ch, 1
```

Il est préférable d'éviter l'utilisation d'identificateurs définis par l'utilisateur portant le même nom qu'un mot réservé de l'assembleur intégré.

Expressions

L'assembleur intégré évalue toutes les expressions en valeurs entières 32 bits. Il ne supporte pas la virgule flottante et les valeurs chaîne, à l'exception des constantes chaîne.

Les expressions sont construites à partir *d'éléments d'expression* et *d'opérateurs* ; chaque expression a une *classe d'expression* et un *type d'expression associé*.

Différences entre les expressions Pascal Objet et assembleur

La différence la plus importante entre les expressions Pascal Objet et celles de l'assembleur intégré est que toutes les expressions de l'assembleur doivent se résoudre en une *valeur constante* ; une valeur qui peut être calculée à l'exécution. Par exemple, soit ces déclarations :

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

L'instruction suivante est correcte :

```
asm
  MOV     Z,X+Y
end;
```

Comme *X* et *Y* sont des constantes, l'expression `X + Y` est juste un moyen commode d'écrire la constante 30 et l'instruction résultante place simplement la valeur 30 dans la variable *Z*. Mais si *X* et *Y* sont des variables :

```
var
  X, Y: Integer;
```

L'assembleur intégré ne peut calculer la valeur de `X + Y` à la compilation. Dans ce cas, pour placer la somme de *X* et *Y* dans *Z*, vous devez utiliser

```
asm
  MOV     EAX,X
  ADD     EAX,Y
  MOV     Z,EAX
end;
```

Dans une expression Pascal Objet, une référence de variable désigne le *contenu* de la variable alors que dans une expression assembleur, une référence de variable désigne *l'adresse* de la variable. En Pascal Objet l'expression `X + 4` (où *X* est une variable) correspond au contenu de X plus 4, alors que pour l'assembleur intégré cela désigne le contenu du mot situé à l'adresse située quatre octets après celle de *X*. Donc, même si vous avez le droit d'écrire :

```
asm
  MOV     EAX,X+4
end;
```

ce code ne charge pas la valeur de *X* plus 4 dans AX : il charge la valeur d'un mot stocké quatre octets après *X*. La manière correcte d'ajouter 4 au contenu de *X* est :

```
asm
  MOV     EAX,X
  ADD     EAX,4
end;
```

Eléments d'expression

Les éléments composant une expression sont *les constantes*, *les registre*s et les *symboles*.

Constantes

L'assembleur intégré supporte deux types de constantes : les *constantes numériques* et les *constantes chaîne*.

Constantes numériques

Les constantes numériques doivent être des entiers et leurs valeurs doivent être comprises entre –2 147 483 648 et 4 294 967 295.

Par défaut, les constantes numériques utilisent la notation décimale, mais l'assembleur intégré supporte également les notations binaires, octales et hexadécimales. La notation binaire est sélectionnée en écrivant un *B* après le nombre, la notation octale en écrivant une lettre *O* après le nombre et la notation hexadécimale en écrivant un *H* après le nombre ou un $ avant le nombre.

Les constantes numériques doivent commencer par un des chiffres compris entre 0 et 9 ou un caractère $. Par conséquent lorsque vous écrivez une constante hexadécimale en utilisant le suffixe *H*, un zéro supplémentaire devant le nombre est nécessaire si le premier chiffre significatif est un des chiffres allant de *A* à *F*. Par exemple, 0BAD4H et $BAD4 sont des constantes hexadécimales, mais BAD4H est un identificateur car il commence par une lettre.

Constantes chaîne

Les constantes chaîne doivent être entourées d'apostrophes ou de guillemets. Deux apostrophes (ou deux guillemets) consécutives de même type comptent pour un seul caractère. Voici des exemples de constantes chaîne :

```
'Z'
'Delphi'
'Linux'
"That's all folks"
'"That''s all folks," he said.'
'100'
'"'
"'"
```

Les constantes chaîne de toute longueur sont autorisées dans les directives DB et provoquent l'allocation d'une séquence d'octets contenant les valeurs ASCII des caractères de la chaîne. Dans tous les autres cas, une constante chaîne ne peut pas dépasser quatre caractères, et donne une valeur numérique pouvant se trouver dans une expression. La valeur numérique d'une constante chaîne est calculée ainsi :

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

où *Ch1* est le caractère le plus à droite (le dernier) et *Ch4* le plus à gauche (le premier). Si la chaîne est inférieure à quatre caractères, les caractères les plus à gauche sont pris pour zéro.

Le tableau suivant présente certains exemples de constantes chaîne et de leurs valeurs numériques.

Tableau 13.2 Exemples de chaînes et de la valeur correspondante

Chaîne	Valeur
`'a'`	00000061H
`'ba'`	00006261H
`'cba'`	00636261H
`'dcba'`	64636261H
`'a  '`	00006120H
`'   a'`	20202061H
`'a' * 2`	000000E2H
`'a'-'A'`	00000020H
not `'a'`	FFFFFF9EH

Registres

Les symboles réservés suivants indiquent des registres CPU :

Tableau 13.3 Registres de la CPU

Général 32 bits	EAX EBX ECX EDX	Pointeur ou index 32 bits	ESP EBP ESI EDI
Général 16 bits	AX BX CX DX	Pointeur ou index 16 bits	SP BP SI DI
Registres 8 bits poids faible	AL BL CL DL	Registres de segment 16 bits	CS DS SS ES
Registres 8 bits poids fort	AH BH CH DH	Pile de registre de coprocesseur	ST

Lorsqu'une opérande consiste uniquement en un nom de registre, elle est appelée *opérande registre*. Tous les registres peuvent être utilisés comme opérandes registre, certains registres peuvent être utilisés dans d'autres contextes.

Les registres de base (BX et BP) et les registres d'index (SI et DI) peuvent être écrits entre des crochets pour indiquer l'indexation. Les combinaisons de registre de base/d'index correctes sont [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI] et[BP+DI]. Vous pouvez aussi indexer avec tous les registres 32 bits, par exemple [EAX+ECX], [ESP] et [ESP+EAX+5].

Les registres de segment (ES, CS, SS, DS, FS et GS) sont supportés, mais les segments ne sont pas utiles dans les applications 32 bits.

Le symbole ST indique le registre le plus au-dessus de la pile de registre en virgule flottante 8087. ST(*X*) permet de faire référence à chacun des huit registres

en virgule flottante, *X* étant une constante comprise entre 0 et 7 qui indique la distance depuis le haut de la pile de registre.

Symboles

L'assembleur intégré vous permet d'accéder à presque tous les symboles Pascal Objet dans des expressions assembleur, dont les constantes, les types, variables, les procédures et les fonctions. De plus, l'assembleur intégré implémente le symbole spécial *@Result* qui correspond à la variable *Result* à l'intérieur d'une fonction. Ainsi, la fonction suivante :

```
function Sum(X, Y: Integer): Integer;
begin
  Result := X + Y;
end;
```

peut s'écrire en assembleur de la manière suivante :

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    ADD     EAX,Y
    MOV     @Result,EAX
  end;
end;
```

Il n'est pas permis d'utiliser les symboles suivants dans des instructions **asm** :

- Les procédures et fonctions standard (par exemple, *Writeln* ou *Chr*)
- Les tableaux spéciaux *Mem*, *MemW*, *MemL*, *Port* et *PortW*
- Les constantes chaîne, virgule flottante et ensemble
- Les étiquettes non déclarées dans le bloc en cours
- Le symbole *@Result* hors d'une fonction

Le tableau suivant résume les types de symbole utilisables dans les instructions **asm**.

Tableau 13.4 Symboles reconnus par l'assembleur intégré

Symbole	Valeur	Classe	Type
Label	Adresse d'étiquette	Mémoire	SHORT
Constant	Valeur de constante	Immédiat	0
Type	0	Mémoire	Taille du type
Field	Déplacement de champ	Mémoire	Taille du type
Variable	Adresse de variable	Mémoire	Taille du type
Procedure	Adresse de procédure	Mémoire	NEAR
Function	Adresse de fonction	Mémoire	NEAR
Unit	0	Immédiat	0
@Code	Adresse de segment de code	Mémoire	0FFF0H
@Data	Adresse de segment de données	Mémoire	0FFF0H
@Result	Déplacement de la variable de résultat	Mémoire	Taille du type

Si l'optimisation est désactivée, les variables locales (variables déclarées dans des procédures et fonctions) sont toujours allouées sur la pile et accessibles relativement à EBP et la valeur d'un symbole de variable locale est son déplacement signé depuis EBP. L'assembleur ajoute automatiquement [EBP] dans les références aux variables locales. Par exemple, soit ces déclarations :

```
var Count: Integer;
```

dans une fonction ou une procédure, l'instruction :

```
MOV     EAX,Count
```

s'assemble en MOV EAX,[EBP–4].

L'assembleur intégré traite les paramètres **var** comme des pointeurs 32 bits et la taille d'un paramètre **var** est toujours 4. La syntaxe pour accéder à un paramètre **va**r est différente de celle utilisée pour accéder à un paramètre transmis par valeur. Pour accéder au contenu d'un paramètre **var**, vous devez d'abord charger le pointeur 32 bits puis ensuite accéder à l'emplacement qu'il désigne. Par exemple :

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    MOV     EAX,[EAX]
    MOV     EDX,Y
    ADD     EAX,[EDX]
    MOV     @Result,AX
  end;
end;
```

Il est possible de qualifier les identificateurs dans les instructions **asm**. Par exemple, étant donné les déclarations :

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: TPoint;
  end;
var
  P: TPoint;
  R: TRect;
```

Il est possible d'utiliser les constructions suivantes dans une instruction **asm** pour accéder aux champs :

```
MOV     EAX,P.X
MOV     EDX,P.Y
MOV     ECX,R.A.X
MOV     EBX,R.B.Y
```

Un identificateur de type peut être utilisé pour construire des variables à la volée. Chacune des instructions suivantes génère le même code machine, qui charge le contenu de [EDX] dans EAX.

```
MOV     EAX,(TRect PTR [EDX]).B.X
MOV     EAX,TRect(EDX]).B.X
MOV     EAX,TRect[EDX].B.X
MOV     EAX,[EDX].TRect.B.X
```

Classes d'expression

L'assembleur intégré divise les expressions en trois classes : *registres*, *références mémoire* et *valeurs immédiates*.

Une expression qui ne comprend qu'un nom de registre est une expression registre. Des exemples d'expressions registre sont AX, CL, DI et ES. Utilisées comme opérandes, les expressions registre dirigent l'assembleur pour générer des instructions qui fonctionnent sur les registres CPU.

Les expressions qui indiquent des emplacements mémoire sont des références mémoire ; les étiquettes, variables, constantes typées, procédures et fonctions de Pascal Objet appartiennent à cette catégorie.

Les expressions qui ne sont pas des registres et qui ne sont pas associées aux emplacements mémoire sont des valeurs immédiates. Ce groupe inclut les constantes non typées et les identificateurs de type du Pascal Objet.

Les valeurs immédiates et les références mémoire provoquent la génération d'un code différent lorsqu'elles sont utilisées comme opérandes. Par exemple,

```
const
  Start = 10;
var
  Count: Integer;
  ⋮
asm
  MOV     EAX,Start           { MOV EAX,xxxx }
  MOV     EBX,Count           { MOV EBX,[xxxx] }
  MOV     ECX,[Start]         { MOV ECX,[xxxx] }
  MOV     EDX,OFFSET Count    { MOV EDX,xxxx }
end;
```

Comme *Start* est une valeur immédiate, le premier MOV est assemblé dans une instruction de déplacement immédiat. Le deuxième MOV, cependant, est traduit en une instruction mémoire de déplacement, comme *Count* est une référence mémoire. Dans le troisième MOV, l'opérateur crochet est utilisé pour convertir *Start* en référence mémoire (dans ce cas, le mot à l'offset 10 du segment de données), et dans le quatrième MOV, l'opérateur OFFSET est utilisé pour convertir *Count* en une valeur immédiate (l'offset de *Count* dans le segment de données).

Les crochets et les opérateurs OFFSET se complémentent l'un l'autre. L'instruction **asm** suivante est identique aux deux premières lignes de l'instruction **asm** précédente :

```
asm
  MOV     EAX,OFFSET [Start]
  MOV     EBX,[OFFSET Count]
end;
```

Les références mémoire et les valeurs immédiates sont classées comme *relogeables* ou *absolues*. Le relogement est le processus par lequel le lieur affecte des adresses absolues aux symboles. Une expression relogeable désigne une valeur qui nécessite un relogement lors de la liaison alors qu'une expression absolue désigne une valeur qui n'a pas besoin de relogement. Habituellement, les expressions faisant référence à des labels, des variables, des procédures ou des fonctions sont relogeables car l'adresse finale de ces symboles est inconnue lors de la compilation. Les expressions qui agissent uniquement sur des constantes sont absolues.

L'assembleur intégré vous permet d'exécuter toute opération sur une valeur absolue, mais restreint les opérations sur les valeurs relogeables pour l'addition et la soustraction des constantes.

Types d'expression

Chaque expression de l'assembleur intégré a un type ou plus précisément une taille associée, car l'assembleur intégré considère le type d'une expression simplement comme la taille de son emplacement mémoire. Par exemple, le type d'une variable *Integer* est quatre car il occupe 4 octets. L'assembleur intégré effectue des vérifications de type à chaque fois que cela est possible, comme dans les instructions suivantes :

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  ⋮
asm
  MOV     AL,QuitFlag
  MOV     BX,OutBufPtr
end;
```

où l'assembleur vérifie que la taille de *QuitFlag* est un (un octet) et que la taille de *OutBufPtr* est deux (un mot). L'instruction :

```
MOV     DL,OutBufPtr
```

produit une erreur car DL est un registre de taille octet et *OutBufPtr* est un mot. Le type d'une référence mémoire peut être changé via un transtypage ; ainsi pour écrire correctement l'instruction précédente, vous pouvez utiliser :

```
MOV     DL,BYTE PTR OutBufPtr
MOV     DL,Byte(OutBufPtr)
MOV     DL,OutBufPtr.Byte
```

Ces instructions MOV font toutes référence au premier octet (le moins significatif) de la variable *OutBufPtr*.

Dans certains cas, une référence mémoire est non typée ; c'est-à-dire qu'elle n'a pas de type associé. Un exemple est une valeur immédiate entourée de crochets :

```
MOV     AL,[100H]
MOV     BX,[100H]
```

L'assembleur intégré permet ces deux instructions, car l'expression [100H] n'a pas de type. Cela signifie seulement "le contenu de l'adresse 100H dans le segment de données," et le type peut être déterminé à partir de la première opérande (octet pour AL, mot pour BX). Dans les cas où le type ne peut pas être déterminé à partir d'une autre opérande, l'assembleur intégré requiert un transtypage explicite :

```
INC     BYTE PTR [100H]
IMUL    WORD PTR [100H]
```

Le tableau suivant résume les symboles de type prédéfini que l'assembleur intégré fournit en plus de tout type Pascal Objet actuellement déclaré.

Tableau 13.5 Symboles de type prédéfinis

Symbole	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Opérateurs d'expression

L'assembleur intégré propose divers opérateurs. Les règles de priorité sont différentes de celles s'appliquant en Pascal Objet. Par exemple, dans une instruction **asm**, AND a une priorité plus faible que les opérateurs d'addition et de soustraction. Le tableau suivant énumère les opérateurs d'expression de l'assembleur intégré en ordre décroissant de priorité.

Tableau 13.6 Priorité des opérateurs d'expression de l'assembleur intégré

Opérateurs	Remarque	Priorité
&		Maximum
(), **[]**, **.**, **HIGH**, **LOW**		
+, **–**	**+** et **–** **unaires**	
:		
OFFSET, **SEG**, **TYPE**, **PTR**, *****, **/**, **MOD**, **SHL**, **SHR**, **+**, **–**	**+** et **–** **binaires**	
NOT, **AND**, **OR**, **XOR**		Minimum

Le tableau suivant définit les opérateurs d'expression de l'assembleur intégré.

Tableau 13.7 Définitions des opérateurs d'expression de l'assembleur intégré

Opérateur	Description
&	**Surcharge d'identificateur**. L'identificateur suivant immédiatement cet opérateur est traité comme symbole défini par l'utilisateur, même si l'orthographe est identique à celle du symbole réservé de l'assembleur intégré.
(...)	**Sous-expression**. Les expressions entre parenthèses sont évaluées complètement avant d'être traitées comme élément d'expression simple. Une autre expression peut parfois précéder l'expression entre parenthèses ; le résultat dans ce cas devient la somme des valeurs des deux expressions, avec le type de la première expression.
[...]	**Référence mémoire**. L'expression entre crochets est évaluée complètement avant d'être traitée comme élément d'expression simple. L'expression entre crochets peut être combinée avec les registres BX, BP, SI ou DI en utilisant l'opérateur plus (+) pour indiquer l'indexation de registre CPU. Une autre expression peut parfois précéder l'expression entre crochets ; le résultat dans ce cas devient la somme des valeurs de deux expressions, avec le type de la première expression. Le résultat est toujours une référence mémoire.
.	**Sélecteur de membre de structure**. Le résultat est la somme de l'expression située avant le point avec celle située après le point avec le type de l'expression située après le point. L'accès aux symboles appartenant à la portée identifiée par l'expression située avant le point peut s'effectuer dans l'expression située après le point.
HIGH	Renvoie les huit bits de poids fort de l'expression mot suivant l'opérateur. L'expression doit être une valeur immédiate absolue.
LOW	Renvoie les huit bits de poids faible de l'expression mot suivant l'opérateur. L'expression doit être une valeur immédiate absolue.
+	**Plus unaire**. Renvoie l'expression suivant le plus sans changement. L'expression doit être une valeur immédiate absolue.
–	**Moins unaire**. Renvoie la valeur négative de l'expression suivant le moins. L'expression doit être une valeur immédiate absolue.
+	**Addition**. Les expressions peuvent être des valeurs immédiates ou des références mémoire, mais une seule des expressions peut être une valeur relogeable. Si une des expressions est une valeur relogeable, le résultat est aussi une valeur relogeable. Si les deux expressions sont des références mémoire, le résultat est aussi une référence mémoire.
–	**Soustraction**. La première expression peut avoir n'importe quelle classe, mais la deuxième expression doit être une valeur immédiate absolue. Le résultat a la même classe que la première expression.
:	**Surcharge de segment**. Instruit l'assembleur que l'expression située après les deux-points appartient au segment donné par le nom de registre segment (CS, DS, SS, FS, GS ou ES) situé avant les deux-points. Le résultat est une référence mémoire avec la valeur de l'expression située après les deux-points. Lorsqu'une surcharge de segment est utilisée dans une opérande d'instruction, l'instruction aura comme préfixe une instruction préfixe de surcharge de segment appropriée. Ceci pour s'assurer que le segment indiqué est sélectionné.
OFFSET	Renvoie la partie offset (double mot) de l'expression suivant l'opérateur. Le résultat est une valeur immédiate.
TYPE	Renvoie le type (taille en octets) de l'expression suivant l'opérateur. Le type d'une valeur immédiate est 0.

Tableau 13.7 Définitions des opérateurs d'expression de l'assembleur intégré (suite)

Opérateur	Description
PTR	**Opérateur de transtypage**. Le résultat est une référence mémoire avec la valeur de l'expression suivant l'opérateur et le type de celle située devant l'opérateur.
*	**Multiplication**. Les deux expressions doivent être des valeurs immédiates absolues et le résultat est une valeur immédiate absolue.
/	**Division entière**. Les deux expressions doivent être des valeurs immédiates absolues et le résultat est une valeur immédiate absolue.
MOD	**Reste après la division entière**. Les deux expressions doivent être des valeurs immédiates absolues et le résultat est une valeur immédiate absolue.
SHL	**Décalage gauche logique**. Les deux expressions doivent être des valeurs immédiates absolues et le résultat est une valeur immédiate absolue.
SHR	**Décalage droite logique**. Les deux expressions doivent être des valeurs immédiates absolues et le résultat est une valeur immédiate absolue.
NOT	**Négation bit-à-bit.** L'expression doit être une valeur immédiate absolue et le résultat est une valeur immédiate absolue.
AND	**AND bit-à-bit**. Les deux expressions doivent être des valeurs immédiates absolues et le résultat est une valeur immédiate absolue.
OR	**OR bit-à-bit**. Les deux expressions doivent être des valeurs immédiates absolues et le résultat est une valeur immédiate absolue.
XOR	**OR exclusif bit-à-bit**. Les deux expressions doivent être des valeurs immédiates absolues et le résultat est une valeur immédiate absolue.

Procédures et fonctions assembleur

Il est possible d'écrire entièrement en code assembleur en ligne des procédures ou des fonctions sans inclure d'instruction **begin...end**. Par exemple :

```
function LongMul(X, Y: Integer): Longint;
asm
  MOV     EAX,X
  IMUL    Y
end;
```

Le compilateur effectue plusieurs optimisations sur ces routines :

- Il n'y a pas de code généré pour copier des paramètres par valeur dans des variables locales. Ceci affecte tous les paramètres par valeur de type chaîne et les autres paramètres par valeur dont la taille n'est pas de 1, 2 ou 4 octets. Dans la routine, de tels paramètres doivent être traités comme s'il s'agissait de paramètres **var**.
- A moins qu'une fonction ne renvoie une chaîne, un variant ou une référence d'interface, le compilateur n'alloue pas de variable résultat de fonction : une référence au symbole *@Result* est donc une erreur. Pour les chaînes, les variants et les interfaces, l'appelant peut toujours allouer un pointeur *@Result*.
- Le compilateur ne génère pas de cadre de pile pour les routines qui ne sont pas imbriquées et n'ont ni paramètres, ni variables locales.

- Le code d'entrée généré automatiquement pour la routine a la forme suivante :

```
PUSH    EBP             ;Présent si Locals <> 0 ou Params <> 0
MOV     EBP,ESP         ;Présent si Locals <> 0 ou Params <> 0
SUB     ESP,Locals      ;Présent si Locals <> 0
 ⋮
MOV     ESP,EBP         ;Présent si Locals <> 0
POP     EBP             ;Présent si Locals <> 0 ou Params <> 0
RET     Params          ;Toujours présent
```

  Si dans les variables locales il y a des variants, des chaînes longues ou des interfaces, ils sont initialisés à zéro mais pas finalisés.

- *Locals* indique la taille des variables locales et *Params* la taille des paramètres. Si *Locals* et *Params* sont nuls, il n'y a pas de code d'entrée et le code de sortie se limite à une instruction RET.

Les fonctions assembleur renvoient leur résultat de la manière suivante :

- Les valeurs scalaires sont renvoyées dans AL (valeurs sur 8 bits), AX (valeurs sur 16 bits) ou EAX (valeurs sur 32 bits).
- Les valeurs réelles sont renvoyées dans ST(0) dans la pile de registre du coprocesseur. Les valeurs *Currency* sont multipliées par 10000.
- Les pointeurs, y compris les chaînes longues, sont renvoyés dans EAX.

Les chaînes courtes et les variants sont renvoyés dans un emplacement temporaire pointé par *@Result*.

Annexe

Grammaire du Pascal Objet

```
But -> (Programme | Paquet | Bibliothèque | Unité)

Programme -> [PROGRAM Ident ['(' ListeIdent ')'] ';']
           BlocProgramme '.'

Unité -> UNIT Ident ';'
        SectionInterface
        SectionImplémentation
        SectionInit '.'

Paquet -> PACKAGE Ident ';'
           [ClauseRequires]
           [ClauseContains]
           END '.'

Bibliothèque -> LIBRARY Ident ';'
           BlocProgramme '.'

BlocProgramme -> [ClauseUses]
                Bloc

ClauseUses -> USES ListeIdent ';'

SectionInterface -> INTERFACE
                    [ClauseUses]
                    [DéclInterface]...

DéclInterface -> SectionConst
              -> SectionType
              -> SectionVar
              -> SectionExportation

EnTêteExportation -> EnTêteProcédure ';' [Directive]
                  -> EnTêteFonction ';' [Directive]

SectionImplémentation -> IMPLEMENTATION
                         [ClauseUses]
                         [SectionDécl]...
```

```
Bloc -> [SectionDécl]
        InstructionComposée

SectionDécl -> SectionDéclLabel
            -> SectionConst
            -> SectionType
            -> SectionVar
            -> SectionDéclProc

SectionDéclLabel -> LABEL IdentLabel

SectionConst -> CONST (DéclConstant ';')...

DéclConstant -> Ident '=' ExprConst
             -> Ident ':' IdentType '=' ConstanteTypée

SectionType -> TYPE (DéclType ';')...

DéclType -> Ident '=' Type
         -> Ident '=' TypeRestreint

ConstanteTypée -> (ExprConst | ConstanteTableau | ConstanteEnregistrement)

ConstanteTableau -> '(' ConstanteTypée/','... ')'

ConstanteEnregistrement -> '(' ConstChampEnreg/';'... ')'

ConstChampEnreg -> Ident ':' ConstanteTypée

Type -> IdentType
     -> TypeSimple
     -> TypeStruc
     -> TypePointeur
     -> TypeChaîne
     -> TypeProcédure
     -> TypeVariant
     -> TypeRefClass

TypeRestreint -> TypeObjet
              -> TypeClasse
              -> TypeInterface

TypeRefClass -> CLASS OF IdentType

TypeSimple -> (TypeScalaire | TypeRéel)

TypeRéel -> REAL48
         -> REAL
         -> SINGLE
         -> DOUBLE
         -> EXTENDED
         -> CURRENCY
         -> COMP

TypeScalaire -> (TypeIntervalle | TypeEnum | IdentScalaire)

IdentScalaire -> SHORTINT
         -> SMALLINT
         -> INTEGER
         -> BYTE
         -> LONGINT
         -> INT64
         -> WORD
```

```
         -> BOOLEAN
         -> CHAR
         -> WIDECHAR
         -> LONGWORD
         -> PCHAR

TypeVariant -> VARIANT
            -> OLEVARIANT

TypeIntervalle -> ExprConst '..' ExprConst

TypeEnum -> '(' ElémentTypeEnum/','... ')'

ElementTypeEnum -> Ident [ '=' ExprConst ]

TypeChaîne -> STRING
           -> ANSISTRING
           -> WIDESTRING
           -> STRING '[' ExprConst ']'

TypeStruc -> [PACKED] (TypeTableau | TypeEnsemble | TypeFichier | TypeEnreg)

TypeTableau -> ARRAY ['[' TypeScalaire/','... ']'] OF Type

TypeEnreg -> RECORD [ListeChamp] END

ListeChamp ->  DéclChamp/';'... [SectionVariable] [';']

DéclChamp -> ListeIdent ':' Type

SectionVariable -> CASE [Ident ':'] IdentType OF EnregVariable/';'...

EnregVariable -> ExprConst/','...  ':' '(' [ListeChamp] ')'

TypeEnsemble -> SET OF TypeScalaire

TypeFichier -> FILE OF IdentType

TypePointeur -> '^' IdentType

TypeProcédure -> (EnTêteProcédure | EnTêteFonction) [OF OBJECT]

SectionVar -> VAR (DéclVar ';')...

DéclVar -> ListeIdent ':' Type [(ABSOLUTE (Ident | ExprConst)) | '=' ExprConst]

Expression -> ExpressionSimple [OpRel ExpressionSimple]...

ExpressionSimple -> ['+' | '-'] Terme [OpAdd Terme]...

Terme -> Facteur [OpMul Facteur]...

Facteur -> Désignateur ['(' ListeExpr ')']
       -> '@' Désignateur
       -> Nombre
       -> Chaîne
       -> NIL
       -> '(' Expression ')'
       -> NOT Facteur
       -> ConstructeurEnsemble
       -> IdentType '(' Expression ')'

OpRel -> '>'
      -> '<'
      -> '<='
      -> '>='
```

```
        -> '<>'
        -> IN
        -> IS
        -> AS

OpAdd   -> '+'
        -> '-'
        -> OR
        -> XOR

OpMul   -> '*'
        -> '/'
        -> DIV
        -> MOD
        -> AND
        -> SHL
        -> SHR

Désignateur -> IdentQualif ['.' Ident | '[' ListeExpr ']' | '^']...

ConstructeurEnsemble -> '[' [ElémentEnsemble/','...] ']'

ElémentEnsemble -> Expression ['..' Expression]

ListeExpr -> Expression/','...

Instruction -> [IdentLabel ':'] [InstructionSimple | InstructionStructurée]

ListeInstructions -> Instruction/';'...

InstructionSimple -> Désignateur ['(' ListeExpr ')']
                  -> Désignateur ':=' Expression
                  -> INHERITED
                  -> GOTO IdentLabel

InstructionStructurée -> InstructionComposée
           -> InstructionCondition
           -> InstructionBoucle
           -> InstructionWith

InstructionComposée -> BEGIN ListeInstructions END

InstructionCondition -> InstructionIf
                  -> InstructionCase

InstructionIf -> IF Expression THEN Instruction [ELSE Instruction]

InstructionCase -> CASE Expression OF SélecteurCase/';'... [ELSE ListeInstructions] [';']
END

SélecteurCase -> LabalCase/','... ':' Instruction

LabalCase -> ExprConst ['..' ExprConst]

InstructionBoucle -> InstructionRepeat
         -> InstructionWhile
         -> InstructionFor

InstructionRepeat -> REPEAT Instruction UNTIL Expression

InstructionWhile -> WHILE Expression DO Instruction

InstructionFor -> FOR IdentQualif ':=' Expression (TO | DOWNTO) Expression DO Instruction

InstructionWith -> WITH ListeIdent DO Instruction
```

```
SecDéclProcédure -> DéclProcédure
                    -> DéclFonction

DéclProcédure -> EnTêteProcédure ';' [Directive]
                 Bloc ';'

DéclFonction -> EnTêteFonction ';' [Directive]
                Bloc ';'

EnTêteFonction -> FUNCTION Ident [ParamètresFormels] ':' (TypeSimple | STRING)

EnTêteProcédure -> PROCEDURE Ident [ParamètresFormels]

ParamètresFormels -> '(' ParamètreFormel/';'... ')'

ParamètreFormel -> [VAR | CONST | OUT] Paramètre

Paramètre -> ListeIdent [':' ([ARRAY OF] TypeSimple | STRING | FILE)]
          -> Ident ':' TypeSimple '=' ExprConst

Directive -> CDECL
          -> REGISTER
          -> DYNAMIC
          -> VIRTUAL
          -> EXPORT
          -> EXTERNAL
          -> FAR
          -> FORWARD
          -> MESSAGE
          -> OVERRIDE
          -> OVERLOAD
          -> PASCAL
          -> REINTRODUCE
          -> SAFECALL
          -> STDCALL

TypeObjet -> OBJECT [HéritageObjet] [ListeChampObjet] [ListeMéthode] END

HéritageObjet -> '(' IdentQualif ')'

ListeMéthode -> (EnTêteMéthode [';' VIRTUAL])/';'...

EnTêteMéthode -> EnTêteProcédure
              -> EnTêteFonction
              -> EnTêteConstructeur
              -> EnTêteDestructeur

EnTêteConstructeur -> CONSTRUCTOR Ident [ParamètresFormels]

EnTêteDestructeur -> DESTRUCTOR Ident [ParamètresFormels]

ListeChampObjet -> (ListeIdent ':' Type)/';'...

InitSection -> INITIALIZATION ListeInstructions [FINALIZATION ListeInstructions] END
            -> BEGIN ListeInstructions END
            -> END

TypeClasse -> CLASS [HéritageClasse]
             [ListeChampClasse]
             [ListeMéthodeClasse]
             [ListePropriétéClasse]
             END
```

HéritageClasse -> '(' ListeIdent ')'

VisibilitéClasse -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]

ListeChampClasse -> (**VisibilitéClasse** ListeChampObjet)/';'...

ListeMéthodeClasse -> (**VisibilitéClasse** ListeMéthode)/';'...

ListePropriétéClasse -> (**VisibilitéClasse** ListePropriété ';')...

ListePropriété -> PROPERTY Ident [InterfacePropriété] SpécificateursPropriété

InterfacePropriété -> [ListesParamètrePropriété] ':' Ident

ListesParamètrePropriété -> '[' (ListeIdent ':' TypeIdent)/';'... ']'

SpécificateursPropriété -> [INDEX ExprConst]
[READ Ident]
[WRITE Ident]
[STORED (Ident | Constante)]
[(DEFAULT ExprConst) | NODEFAULT]
[IMPLEMENTS IdentType]

TypeInterface -> INTERFACE [HéritageInterface]
[ListeMéthodeClasse]
[ListePropriétéClasse]
END

HéritageInterface -> '(' ListeIdent ')'

ClauseRequires -> REQUIRES ListeIdent... ';'

ClauseContains -> CONTAINS ListeIdent... ';'

ListeIdent -> Ident/','...

IdentQualif -> [IdentUnité '.'] Ident

IdentType -> [IdentUnité '.'] <identificateur de type>

Ident -> <identificateur>

ExprConst -> <expression constante>

IdentUnité -> <identificateur d'unité>

IdentLabel -> <identificateur de label>

Nombre -> <nombre>

Chaîne -> <chaîne>

Index

Symboles

- 4-4, 4-7, 4-10, 4-11
" 13-8
4-5
$ 4-4, 4-6
(*, *) 4-6
(,) 4-2, 4-14, 4-16, 5-6, 5-47, 6-2, 6-3, 6-10, 7-2, 10-1
* 4-2, 4-7, 4-11
+ 4-4, 4-7, 4-10, 4-11
, 3-6, 5-6, 5-23, 6-10, 7-18, 9-5, 9-10, 13-2
. 4-2, 4-3, 4-15, 5-29, 9-10, 10-6
/ 4-2, 4-7
// 4-6
: 4-2, 4-20, 4-26, 5-23, 5-24, 5-41, 5-46, 6-3, 6-10, 7-18, 7-21, 7-31, 13-2
:= 4-20, 4-29
paramètres nommés 10-13
; 3-2, 3-6, 4-18, 4-19, 4-22, 4-28, 5-23, 5-24, 5-41, 6-2, 6-3, 6-10, 7-2, 7-7, 9-5, 9-10, 10-1, 10-4, 10-11
avant 'else' 4-25
< 4-2, 4-10, 4-12
<= 4-10, 4-11, 4-12
<> 4-10, 4-11, 4-12
= 4-2, 4-10, 4-11, 4-12, 4-20, 5-40, 5-44, 5-46, 6-10, 10-6
> 4-2, 4-10, 4-12
>= 4-10, 4-11, 4-12
@ 4-7, 4-13, 5-28, 5-33, 5-48, 7-19
@@ 5-33
@Result 13-16
[,] 4-15, 4-16, 5-11, 5-12, 5-19, 5-34, 6-14, 6-15, 6-20, 7-18, 7-21, 7-22, 10-1, 10-11
^ 4-7, 4-10, 5-21, 5-29
et variants 5-36
présentation des pointeurs 5-28
_ 4-2
{, } 4-6, 10-1, 10-11
' 4-5, 9-10, 10-1, 10-11

A

$A, directive 11-8, 11-9
absolute (directive) 5-42
abstraites
méthodes 7-13
Add, méthode (TCollection) 7-10
addition 4-7
pointeurs 4-10
Addr, fonction 5-29
_AddRef, méthode 10-2, 10-5, 10-10
adresse, opérateur 4-13
adresses absolues 5-42
aide contextuelle (gestion des erreurs) 7-35
alignement (données) 5-18, 11-8
Voir aussi formats de données internes
AllocMemCount, variable 11-2
AllocMemSize, variable 11-2
ancêtres 7-3
and 4-8, 4-9
ANSI, caractères 5-5, 5-13, 5-14
AnsiChar, type 5-5, 5-12, 5-14, 5-30, 11-3
AnsiString, type 5-11, 5-13, 5-14, 5-16, 5-30
Voir aussi chaînes longues
gestion de la mémoire 11-6
tableaux variant et 5-37
appels de fonctions et de procédures récursifs 6-1, 6-4
appels de routines dans 9-1
Append, procédure 8-2, 8-4, 8-6, 8-7
application
partitionnement 9-9
Application, variable 2-6, 3-3
applications 16 bits (compatibilité ascendante) 6-6
applications console 8-4
applications multithreads 5-43
bibliothèques à chargement dynamique 9-7
as 4-13, 7-27, 10-11
ASCII 4-1, 4-5, 5-14
asm, instructions 13-2, 13-16
assembler (directive) 6-6, 13-2
assembleur
routines externes 6-7
assembleur en ligne
code 13-1—13-17
assembleur intégré 13-1—13-17
Assert, procédure 7-29
assertions 7-29
Assign, procédure personnalisée 8-5
Assigned, fonction 5-33, 10-10
AssignFile, procédure 8-2, 8-4, 8-7
at (mot réservé) 7-30
Automation 7-6
Voir aussi COM
appels de méthode 10-13
et variants 11-13
automation 10-11—10-14
interfaces doubles 10-14
automatisables
types 10-12
avancées, déclarations
interfaces 10-4
avancées,déclarations 7-7

B

$B, directive 4-9
begin (mot réservé) 3-3, 4-22, 6-2, 6-3
bibliothèques à chargement dynamique 6-7, 9-1—9-13
chaînes longues 9-8
chargement statique 9-2
écriture 9-4
exceptions 9-8
tableaux dynamiques 9-8
variables 9-1
variables dynamiques 9-8
variables globales 9-7
bibliothèques à chargement statique 9-2
bibliothèques dynamiques *Voir* DLL
bibliothèques *Voir* DLL
blancs 4-1
BlockRead, procédure 8-4
BlockWrite, procédure 8-4
blocs 4-30—4-31
bibliothèque 9-6
fonction 3-4, 6-1, 6-3
intérieur et extérieur 4-32
portée 4-30—4-32
procédure 3-4, 6-1, 6-2
programme 3-1, 3-3
try...except 7-30, 7-33
try...finally 7-34

Boolean, type 5-6, 11-3
booléens
    types 11-3
booléens, types 5-6
BORLANDMM.DLL 9-9
boucle (types scalaires) 5-5
boucles de contrôle 4-22, 4-27
Break, procédure 4-27
    dans le bloc try...finally 7-35
    gestionnaires d'exception 7-32
BSTR, type (COM) 5-14
Byte, type 5-4, 11-3
    assembleur 13-14
ByteArray, type 5-30
ByteBool, type 5-6, 11-3

C

C++ 6-7, 10-1, 11-11
cacher les membres de classes
    Voir aussi méthodes surchargées
caractère
    types 5-5
caractère de fin de ligne 4-1, 8-3
caractères
    littéraux chaîne 4-5, 5-5
    longs 11-3
    pointeurs 5-30
    types 11-3
caractères alphanumériques 4-1, 4-2
caractères de contrôle 4-1, 4-5
Cardinal, type 5-4
case (mot réservé) 4-26, 5-24
-cc, option de ligne de commande du compilateur 8-4
cdecl (convention d'appel) 6-5, 12-2
    constructeurs et destructeurs 12-4
    Self 12-4
    varargs 6-7
chaîne
    Voir aussi jeux de caractères
    paramètres 6-14
chaînes
    comparaison 4-12, 5-11
    constantes 4-5, 5-48, 13-8
    dans les paramètres tableau ouvert 6-17
    étendues 8-8
    gestion de la mémoire 11-6, 11-7
    index 4-16
    littéraux 4-5
    longues 11-3
    opérateurs 4-10, 5-16
    types 5-11—5-17
    variants 5-34
chaînes à zéro terminal 5-14—5-17, 5-30, 11-6, 11-7
    mélange avec des chaînes Pascal 5-16
    routines standard 8-7, 8-8
chaînes compactées 5-20
    comparaison 4-12
chaînes courtes 5-3, 5-11, 5-12
chaînes de caractères 4-1, 4-5, 5-48
chaînes de contrôle 4-5
chaînes de ressource 5-45
chaînes délimitées 4-5, 5-48
chaînes délimitées par des guillemets
    assembleur 13-8
chaînes et caractères étendus 5-14
chaînes et caractères longs
    gestion de la mémoire 11-3
chaînes longues 4-10, 5-11, 5-13
    dans les bibliothèques à chargement dynamique 9-8
    enregistrements et 5-25
    fichiers et 5-27
    gestion de la mémoire 11-3, 11-6
chaînes vides 4-5
champs 5-23—5-26, 7-1, 7-8
    publication 7-6
Char type 5-5
Char, type 5-14, 5-30, 11-3
chemin de recherche des bibliothèques 3-7
chemins de répertoire
    dans la clause uses 3-6
Chr, fonction 5-5
classe
    types 7-1, 7-5
classes 7-1—7-35
    comparaison 4-13
    compatibilité 7-4, 10-10
    déclaration des types classe 7-2, 7-5, 7-7, 7-8, 7-9, 7-18, 10-6
    fichiers et 5-27
    mémoire 11-11
    métaclasses 7-25
    méthodes de classe 7-1, 7-28
    opérateurs 4-13, 7-27
    portée 4-31
    références de classe 7-25
    types classe 7-1, 7-2
    variants et 5-33
classes mutuellement dépendantes 7-7
Classes, unité 7-10, 7-26
ClassParent, méthode 7-27
ClassType, méthode 7-27
clauses de résolution de méthode 10-6, 10-7
clients 3-4
Close, fonction 8-5, 8-7
CloseFile, fonction 8-6
CloseFile, procédure 8-7
CLX 1-2
CmdLine, variable 9-7
codes opératoires (assembleur) 13-2, 13-3
COM 10-4
    et variants 5-34, 5-36, 11-12
    interfaces 10-3
    paramètres out 6-13
COM *Voir aussi* Automation
COM, gestion des erreurs 6-5
commentaires 4-1, 4-6
ComObj, unité 7-6, 10-12
Comp, type 5-10, 11-5
comparaison
    chaînes 4-12, 5-11
    chaînes compactées 4-12
    classes 4-13
    objets 4-13
    opérateurs relationnels 4-12
    PChar, type 4-13
    tableaux dynamiques 5-21
    types entier 4-12
    types réels 4-12
    types référence de classe 4-13
compatibilité des affectations 10-10
compatibilité pour l'affectation 5-39, 7-4
compilateur 2-2, 2-3, 2-6
    ligne de commande 2-3—2-6
    paquets 9-12
compilation
    directives 4-6
composants, des classes *Voir* membres
comptage de références 5-13, 10-10
compteur de référence 11-6, 11-8
concaténation (chaînes) 4-10

conditionnels imbriqués 4-25
conflits de nom 4-32
conflits de nomenclature 3-6
conjonction 4-8
const (mot réservé) 5-44, 5-46, 6-11, 6-13, 6-17, 12-1
constante
  paramètres 12-1
constantes 4-6, 5-43
  assembleur 13-8
  compatibilité de type 5-44
  déclarées 5-43—5-48
  enregistrement 5-47
  pointeur 5-48
  tableau 5-46
  typées 5-46
  vraies 5-44
constantes numériques *Voir* nombres
constantes procédure 5-48
constructeurs 7-1, 7-10, 7-14
  conventions d'appel 12-4
  exceptions 7-31, 7-35
  références de classe et 7-26
constructeurs tableau ouvert 6-17, 6-20
contains, clause 9-10, 9-11
Continue, procédure 4-27
  dans le bloc try...finally 7-35
  gestionnaires d'exception 7-32
contrôle (programmes) 12-1—12-6
contrôle de programme 6-19
conventions d'appel 5-31, 6-5, 12-1
  bibliothèques partagées 9-4
  interfaces 10-3, 10-7
  méthodes 12-4
  spécificateurs d'accès 6-5, 7-19
  varargs, directive 6-7
conventions typographiques 1-2
conversion
  Voir aussi transtypages
  variants 5-34, 5-34—5-36
conversions de types
  types énumérés 5-8
Copy, fonction 5-21
CORBA
  et variants 5-34
  interfaces 10-4
  paramètres out 6-13
corps (routine) 6-1
coupe-feu d'exceptions 6-5
CPU *Voir* registres
Create, méthode 7-14
Currency, type 5-10, 5-30, 11-5

D

.dcp, fichiers 2-3, 9-12
.dcu, fichiers 2-3, 3-8, 9-11, 9-12
Dec, procédure 5-3, 5-4
décalage droite (opérateur bit-à-bit) 4-9
décalage gauche (opérateur bit-à-bit) 4-9
déclarations 4-1, 4-18, 4-30
  avancées 10-4
  champ 7-8
  classe 7-2, 7-8, 7-9, 7-18, 10-6
  constante 5-44, 5-46
  fonction 6-1, 6-3
  forward 3-4, 6-6
  implémentation 7-9
  local 6-10
  méthode 7-9
  paquets 9-9
  procédure 6-1, 6-2
  propriétés 7-18, 7-21
  type 5-40
  variable 5-41
déclarations avancées
  classes 7-7
déclarations d'interface 3-4
déclarations de définition 6-6, 7-7, 7-9, 10-4
déclarations forward
  paramètres par défaut 6-19
déclarations interface
  paramètres par défaut 6-19
déclarés, types 5-1
décrémentation des scalaires 5-3, 5-5
default (directive) 7-22, 10-12
DefaultHandler, méthode 7-18
définition
  déclarations 6-6, 7-7, 7-9, 10-4
DefWindowProc, fonction 7-18
délégation (implémentation des interfaces) 10-7
$DENYPACKAGEUNIT, directive 9-13
dépendance
  unités 3-7—3-9
deprecated, directive 4-19
descendants 7-3, 7-5
$DESIGNONLY, directive 9-13
.desk, fichiers 2-3
Destroy, méthode 7-15, 7-16, 7-32
destructeurs 7-1, 7-15, 7-16
  conventions d'appel 12-4
.DFM, fichiers 7-23
.dfm, fichiers 2-2, 2-8, 7-6
différence (ensembles) 4-11
directives 4-1, 4-4
  Voir aussi mots réservés
  assembleur 13-4
  compilation 4-6
  liste 4-4
  ordre 7-9
directives de conseil 4-19
directives des méthodes
  ordre 7-9
disjonction 4-8
  bit-à-bit 4-9
Dispatch, méthode 7-18
dispid (directive) 7-7, 10-2, 10-12
dispinterface 10-11
dispinterface (mot réservé) 10-2
Dispose, procédure 5-21, 5-42, 7-4, 9-8, 11-1, 11-2
distinction minuscules/majuscules 4-1, 4-2, 6-8
  noms d'unités et fichiers 4-2
div 4-7
division 4-7
dlclose 9-2
DLL 9-1—9-9
  appel des routines dans 6-7
  chaînes longues dans 9-8
  chargement dynamique 9-2
  chargement statique 9-2
  écriture 9-4
  exceptions 9-8
  multithread 9-7
  tableaux dynamiques dans 9-8
  variables 9-1
  variables dynamiques dans 9-8
  variables globales 9-7
.DLL, fichiers 6-7, 9-1
DLL_PROCESS_DETACH 9-8
DLL_THREAD_ATTACH 9-8
DLL_THREAD_DETACH 9-8
DLLProc, variable 9-7
dlopen 9-2
dlsym 9-2
do (mot réservé) 4-23, 4-28, 4-29, 7-31

.dof, fichiers 2-3
données
  alignement 11-8
données, alignement 5-18
Double, type 5-10, 11-5
doubles
  interfaces 10-14
downto (mot réservé) 4-29
.dpk, fichiers 2-2, 9-12
.dpr, fichiers 2-2, 3-2, 3-7
.dpu, fichiers 2-3, 3-8, 9-11, 9-12
.drc, fichiers 2-3
.dsk, fichiers 2-3
DWORD, type (assembleur) 13-14
dynamiques
  méthodes 7-11
  tableaux 5-20, 6-15, 11-8
  variables 5-42

E

E (en nombres) 4-4
EAssertionFailed 7-29
écriture seule
  propriétés 7-21
éditeur de bibliothèque de types 10-3
else (mot réservé) 4-24, 4-26, 7-31
end (mot réservé) 3-3, 4-22, 4-26, 5-23, 5-24, 6-2, 6-3, 7-2, 7-31, 7-35, 9-10, 10-1, 10-11, 13-2
enregistrements 4-23, 5-23—5-26
  constantes 5-47
  dans les propriétés 7-6
  mémoire 11-8
  parties variables 5-24—5-26
  portée 4-31, 5-24
  types enregistrement 5-23
  variants et 5-33
enregistrements compactés 11-9
ensemble
  types 5-18
ensemble vide 5-19
ensembles
  constructeurs d'ensemble 4-15
  mémoire 11-7
  opérateurs 4-11
  publication 7-6
  types ensemble 5-18
  variants et 5-33
en-tête
  programme 2-1, 3-1, 3-2
  routine 6-1
  unité 3-3, 3-4
entiers
  types 11-3
entiers, opérateurs 4-7
entiers, types 5-4
énumérés, types 5-6, 11-4
Eof, fonction 8-6
Eoln, fonction 8-6
ErrorAddr, variable 12-6
espaces 4-1
EStackOverflow, exception 11-2
étendues
  chaînes 5-14
étendus
  caractères 5-14
étendus, chaînes et caractères
  routines standard 8-8
évaluation complète 4-8
évaluation optimisée 4-8
évaluation partielle 4-8
EVariantError, exception 5-36
événements 2-8, 7-6
except (mot réservé) 7-31
ExceptAddr, fonction 7-35
Exception, classe 7-29, 7-35
ExceptionInformation, variable 9-8
exceptions 4-22, 7-15, 7-16, 7-28—7-35
  bibliothèques à chargement dynamique 9-6, 9-8
  constructeurs 7-31, 7-35
  dans la section initialisation 7-31
  déclaration 7-29
  déclenchement 7-30
  destruction 7-31, 7-32
  exceptions standard 7-35
  fichier E/S 8-3
  gestion 7-30, 7-31, 7-32, 7-33, 7-35
  imbriquées 7-34
  propagation 7-32, 7-34, 7-35
  redéclenchement 7-33
  routines standard 7-35
exceptions imbriquées 7-34
ExceptObject, fonction 7-35
Exit, procédure 6-2
  dans le bloc try...finally 7-35
  gestionnaires d'exception 7-32
ExitCode, variable 9-6, 12-6
ExitProc, variable 9-6, 9-7, 12-5
export (directive) 6-6
exports, clause 4-30, 9-5
  routines surchargées 9-6
expression constante
  constante, déclarations 5-44
  constantes tableau 5-46
  déclarations de constantes 5-46, 5-47
  définition 5-45
  instructions case 4-26
  intervalle, types 5-8, 5-9
  paramètres par défaut 6-18
  type 5-44, 6-9
  types énumérés 5-8
expressions 4-1, 4-6
  assembleur 13-6—13-16
expressions absolues (assembleur) 13-13
expressions relogeables (assembleur) 13-13
Extended, type 4-8, 5-10, 5-30, 11-5
extérieur, bloc 4-32
external (directive) 6-6, 6-7, 9-1, 9-2

F

False 5-6, 11-3
far (directive) 6-6
fiche principale 2-6
fiches 2-2
fichier
  variables 8-2
fichier E/S 8-1—8-7
  exceptions 8-3
fichier, types 5-26
fichiers
  code source 2-2
  comme paramètres 6-11
  générés 2-2, 2-3, 9-10, 9-12
  initialisation 5-42
  mémoire 11-9
  typés 5-27, 8-2
  types de fichier 5-26, 8-2
  variants et 5-33
fichiers code source 2-2
fichiers d'objets partagés 9-1
  exceptions 9-8
fichiers d'options de projet 2-3
fichiers de configuration de bureau 2-3
fichiers de paquet 2-2
fichiers de ressources 2-2, 2-3
fichiers exécutables 2-3
fichiers fiche 2-2, 2-6, 7-6, 7-23
fichiers non typés 8-2

fichiers objet
appel des routines dans 6-7
fichiers objet partagé 2-3
importation de fonctions 6-7
fichiers paquet 2-3, 9-9, 9-12
fichiers projet 2-2, 3-2, 3-7
fichiers sans type 5-27, 8-4
fichiers texte 8-2, 8-3
fichiers texte, pilotes de périphérique 8-5
fichiers unité 3-3
distinction minuscules/ majuscules 4-2
file (mot réservé) 5-27
FilePos, fonction 8-3
FileSize, fonction 8-3
finalisation, section 3-3, 3-5, 12-5
Finalize, procédure 5-20
finally (mot réservé) 7-35
Flush, fonction 8-5, 8-6
flux (données) 7-6
fonctions 3-4, 6-1—6-20
appels de fonctions 4-15, 4-20, 6-1, 6-19—6-20
appels externes 6-7
assembleur 13-16
déclaration 6-3, 6-6
imbriquées 5-32, 6-10
pointeurs 4-13, 5-31
redéfinition 6-6, 6-8
type de la valeur renvoyée 6-3
type de retour 6-4
valeur renvoyée 6-3, 6-4
valeurs renvoyées dans les registres 12-3, 13-17
fonctions de périphérique 8-6
fonctions et procédures redéfinies 6-6, 6-8
forward, déclarations 6-9
paramètres par défaut 6-9, 6-19
fondamentaux, types 5-2
formats de données internes 11-3—11-13
formels paramètres 6-19
fortement typé 5-1
forward, déclarations
redéfinition et 6-9
routines 3-4, 6-6
Free, méthode 7-16
FreeLibrary, fonction 9-2
FreeMem, procédure 5-42, 9-8, 11-1, 11-2

G

$G, directive 9-12
-$G-, option de compilation 9-13
génériques, types 5-2
gestion d'erreurs *Voir* exceptions
gestion de la mémoire 11-1—11-13
gestionnaire de projet 2-1
gestionnaires d'événements 2-8, 7-6
gestionnaires d'exception 7-28, 7-31
identificateurs dans 7-32
gestionnaires de messages 7-16
héritage 7-18
redéfinition 7-17
get *Voir* spécificateur read
GetHeapStatus, fonction 11-2
GetMem, procédure 5-29, 5-42, 9-8, 11-1, 11-2
GetMemoryManager, procédure 11-2
GetProcAddress, fonction 9-2
GlobalAlloc 11-1
globales
variables 5-41
interfaces 10-10
globaux, identificateurs 4-31
grammaire (formelle) A-1—A-6
gras 1-2
GUID 10-1, 10-3, 10-11

H

$H, directive 5-11, 6-15
Halt, procédure 12-5, 12-6
HelpContext, propriété 7-35
héritage 7-2, 7-3, 7-5
interfaces 10-2
High, fonction 5-3, 5-4, 5-13, 5-20, 5-21, 6-16
HInstance, variable 9-7
$HINTS, directive 4-19

I

$I, directive 8-3
identificateurs 4-1, 4-2, 4-4
dans les gestionnaires d'exception 7-32
globaux et locaux 4-31
portée 4-30—4-32
identificateurs de type 5-2
identificateurs publics (section interface) 3-4
identificateurs qualifiés 3-7, 4-3, 4-32, 5-24
avec Self 7-10
dans les transtypages 4-16, 4-17
pointeurs 5-29
IDispatch 10-10, 10-11
interfaces doubles 10-14
IInterface 10-2
implémentation, section 3-3, 3-4, 3-8
et déclarations forward 6-6
méthodes 7-9
portée 4-31
uses, clause 3-8
implements (directive) 7-24, 10-7
$IMPLICITBUILD, directive 9-12
importation de routines depuis des bibliothèques 9-1
$IMPORTEDDATA, directive 9-12
in (mot réservé) 3-6, 4-11, 5-19, 5-36, 9-10
Inc, procédure 5-3, 5-4
incrémentation des scalaires 5-3, 5-5
index 4-16
chaînes 5-11
index (directive) 6-8
index, spécificateur (Windows seulement) 9-5
indices
dans les paramètres var 5-37, 6-13
tableau 5-20, 5-21, 5-22
tableaux variant 5-37
variants chaîne 5-34
indices de propriétés
tableau 7-21
informations de type à l'exécution *Voir* RTTI
inherited (mot réservé) 7-10, 7-15
conventions d'appel 12-4
gestionnaires de messages 7-18
InheritsFrom, méthode 7-27
initialisation
bibliothèques à chargement dynamique 9-6
fichiers 5-42

objets 7-14
unités 3-5
variables 5-41, 5-42
variants 5-34, 5-42
initialisation, section 3-3, 3-5
exceptions 7-31
Initialize, procédure 5-42
inline (mot réservé) 13-2
InOut, fonction 8-5, 8-6
input (paramètre de program) 3-2
Input, variable 8-3
inspecteur d'objets 7-6
instructions 4-1, 4-19—4-30, 6-1
instructions case 4-26
instructions composées 4-22
instructions conditionnelles 4-22
instructions d'affectation 4-20
transtypages 4-16, 4-17
instructions de boucle 4-22, 4-27
instructions for 4-22, 4-27, 4-29
instructions goto 4-21
instructions if...then 4-24
imbriquées 4-25
instructions repeat 4-22, 4-27, 4-28
instructions simples 4-19
instructions structurées 4-22
instructions try...except 4-22
instructions try...finally 4-22
instructions while 4-22, 4-27, 4-28
instructions with 4-22, 4-23
Int64, type 4-8, 5-3, 5-4, 5-10, 11-3
procédures et fonctions standard 5-4
variants et 5-33
Integer, type 4-8, 5-4
intégrés, types 5-1
interface
déclarations 3-4
interface de répartition
types 10-11
interface déléguée 10-7
interface, section 3-3, 3-4, 3-8
déclarations forward et 6-6
méthodes 7-9
portée 4-31
uses, clause 3-8
interfaces 7-2, 10-1—10-14
accès 10-9—10-11
automation 10-11
compatibilité 10-10
conventions d'appel 10-3
délégation 10-7
enregistrements et 5-25
gestion de la mémoire 11-3
GUID 10-1, 10-3, 10-11
implémentation 10-5—10-8
interrogation 10-11
libération 5-43
propriétés 10-2, 10-4, 10-7
références d'interface 10-9—10-11
résolution de méthode, clauses 10-6, 10-7
transtypages 10-10
types interface 10-1—10-4
types interface de répartition 10-11
interfaces doubles 10-3, 10-14
méthodes 6-6
interfaces objet *Voir* interfaces, COM, CORBA
intérieur, bloc 4-32
internes, formats de données 11-3—11-13
interrogation (interfaces) 10-11
intersection (ensembles) 4-11
intervalle, types 5-8
IntToHex, fonction 5-4
IntToStr, fonction 5-4
Invoke, méthode 10-11
IOResult, fonction 8-3, 8-5
is 4-13, 5-36, 7-27
IsLibrary, variable 9-7
italique 1-2
IUnknown 10-2, 10-5, 10-10
IUnknown *Voir* IInterface

J

$J, directive 5-46
Java 10-1
jeux de caractères
ANSI 5-5, 5-13, 5-14
étendus 5-14
mono-octet (SBCS) 5-14
multi-octets (MBCS) 5-14
Pascal 4-1, 4-2, 4-5
jeux de caractères multi-octets
routines de gestion des chaînes 8-8
jump, instructions (assembleur) 13-3

L

labels 4-1, 4-4, 4-21
assembleur 13-2, 13-3
langage assembleur
assembleur intégré 13-1—13-17
Pascal Objet et 13-1, 13-5, 13-6, 13-7, 13-10, 13-12, 13-14
routines assembleur 13-16
-$LE-, option de compilation 9-13
lecture seule
propriétés 7-21
Length, fonction 5-11, 5-20, 5-21
liaison
champs 7-8
méthodes 7-11
liaison à l'exécution *Voir* méthodes dynamiques, méthodes virtuelles
liaison à la compilation *Voir* méthodes statiques
library (mot réservé) 9-4
library, directive 4-19
littéraux chaîne 5-48
-$LN-, option de compilation 9-13
LoadLibrary, fonction 9-2
local (directive) 9-5
local (directive) (Linux seulement) 9-5
LocalAlloc 11-1
locales
variables 5-41, 6-10
locaux, identificateurs 4-31
LongBool, type 5-6, 11-3
Longint, type 5-4, 11-3
Longword, type 5-4, 11-3
Low, fonction 5-3, 5-4, 5-13, 5-20, 5-21, 6-16
-$LU-, option de compilation 9-13

M

$M, directive 7-5, 7-6
masquer les implémentations d'interface 10-7
masquer les membres de classes 7-8, 7-12, 7-24
reintroduce 7-13
$MAXSTACKSIZE, directive 11-2

membres de classes
automatisés 7-4, 7-6
membres de classes privés 7-4, 7-5
membres de classes
protégés 7-4, 7-5
membres de classes publics 7-4, 7-5
membres de classes publiés 7-4, 7-5
$M, directive 7-6
restrictions 7-6
membres *Voir* ensembles
membres, de classes 7-1
interfaces 10-2
visibilité 7-4
même espace mémoire (dans les enregistrements) 5-25
mémoire 4-1, 5-2, 5-27, 5-28, 5-34, 5-42, 7-16
bibliothèques à chargement dynamique 9-7
gestion 11-1—11-13
gestionnaire de mémoire partagée 9-8
tas 5-42
message (directive) 7-16
interfaces 10-7
Message, propriété 7-35
messages répartition 7-18
Messages, unit 7-17
métaclasses 7-25
méthodes 7-1, 7-2, 7-9—7-18
abstraites 7-13
automation 7-7, 10-13
constructeurs 7-14, 12-4
conventions d'appel 12-4
destructeurs 7-16, 12-4
implémentation 7-9
interface de répartition 10-12
interfaces doubles 6-6
liaison 7-11
méthodes de classe 7-28
pointeurs 4-13, 5-31
publication 7-6
redéfinition 7-11, 7-12
répartition des appels 7-12
statiques 7-11
surcharge 10-6
virtuelles 7-7
méthodes dynamiques 7-11
méthodes redéfinies 7-13
spécificateurs d'accès 7-14, 7-19
méthodes statiques 7-11
méthodes surchargées
publication 7-6
méthodes virtuelles 7-11
automation 7-7
constructeurs 7-15
redéfinition 7-13
$MINSTACKSIZE, directive 11-2
mise en flux (données) 5-2
mod 4-7
modules *Voir* unités
mots réservés 4-1, 4-2, 4-3
Voir aussi directives
assembleur 13-6
liste 4-3
multidimensionnels, tableaux 5-20, 5-22
multi-octets, jeux de caractères 5-14
multiplication 4-7

N

name (directive) 6-8, 9-5
near (directive) 6-6
négation 4-8
bit-à-bit 4-9
New, procédure 5-21, 5-29, 5-42, 7-4, 9-8, 11-1, 11-2
nil 5-29, 5-33, 5-43, 11-5
nombres 4-1, 4-4
assembleur 13-8
comme labels 4-21
labels 4-4
nombres hexadécimaux 4-4
noms
Voir aussi identificateurs
fonctions 6-3, 6-4
identificateurs 4-18
paquets 9-10
paramètres 10-13
programmes 3-2
routines exportées 9-6
unités 3-4, 3-7
non typés
fichiers 8-2
not 4-7, 4-8, 4-9
Null (variants) 5-34, 5-36
null, caractère 5-14, 11-6, 11-7, 11-10
numérique
type 5-44, 6-9

O

objets 4-23, 7-1
Voir aussi classes
comparaison 4-13
fichiers et 5-27
mémoire 11-11
of object 5-31
objets partagés
chargés dynamiquement 9-2
of (mot réservé) 4-26, 5-18, 5-20, 5-27, 5-31, 6-15, 6-17, 7-25
of object (pointeurs de méthode) 5-31
Ole Automation 5-37
OleVariant 5-37
OleVariant, type 5-30, 5-37
on (mot réservé) 7-31
Open, fonction 8-5, 8-6
OpenString 6-15
opérandes 4-6
opérateur adresse 5-28, 5-33, 5-48
propriétés et 7-19
opérateur d'égalité 4-12
opérateur d'inégalité 4-12
opérateur de déréférencement 4-10, 5-21
présentation des pointeurs 5-28
variants et 5-36
opérateur sous-ensemble 4-11
opérateur sur-ensemble 4-11
opérateurs 4-6—4-15
assembleur 13-14
classe 7-27
priorité 4-14, 7-27
opérateurs arithmétiques 4-7, 5-4
opérateurs binaires 4-7
opérateurs bit-à-bit 4-9
opérateurs booléens 4-8
évaluation complète et partielle 4-8
opérateurs caractère 4-10
opérateurs de priorité 4-14
opérateurs logiques 4-9
opérateurs relationnels 4-12
opérateurs unaires 4-7
or 4-8, 4-9
Ord, fonction 5-3
ordre des directives des méthodes 7-9

out (mot réservé) 6-11, 6-13
OutlineError 7-35
output (paramètre de program) 3-2
Output, variable 8-3

P

$P, directive 6-15
packed (mot réservé) 5-18, 11-8
paires de symboles 4-2
PAnsiChar, type 5-14, 5-30
PAnsiString, type 5-30
paquets 9-9—9-13
  chargement dynamique 9-9
  chargement statique 9-9
  clause uses et 9-9
  compilation 9-12
  déclarations 9-9
  directives de compilation 9-12
  options de compilation 9-13
  variables thread 9-10
paquets d'exécution 9-9
paquets de conception 9-9
par défaut
  paramètres 6-18—6-19, 6-20, 10-13
  propriétés 7-22
par position
  paramètres 10-13
par référence (paramètres) 6-11, 6-13, 10-13, 12-1
par valeur (paramètres) 6-11, 10-13, 12-1
paramètres 5-31, 6-2, 6-3, 6-10—6-19
  Voir aussi procédures et fonctions surchargées
  appels de méthode automation 10-13
  chaînes courtes 6-14
  constante 6-13
  contrôle des programmes 12-1
  conventions d'appel 6-5
  fichier 6-11
  formels 6-19
  indices de propriétés tableau 7-21
  liste de paramètres 6-10
  nombre variable 6-7
  output (out) 6-13
  par défaut 6-18—6-19
  propriétés 7-19
  redéfinition et 6-6, 6-8, 6-9
  réels 6-19
  registres 6-5, 12-2
  sans type 6-14, 6-20
  tableau 6-11, 6-15
  tableau ouvert 6-15
  transmission 12-1
  typés 6-11
  valeur 6-11
  variable (var) 6-11
paramètres constante 6-11, 6-13, 6-19, 12-1
  constructeurs tableau ouvert 6-20
paramètres nommés 10-13
paramètres out (output) 6-11, 6-13, 6-19
paramètres par défaut 6-11, 6-20
  déclarations forward et interface 6-19
  objets automation 10-13
  redéfinition et 6-9, 6-19
  types procédure 6-18
paramètres par position 10-13
paramètres tableau ouvert 6-15, 6-20
  et tableaux dynamiques 6-15
paramètres tableau ouvert variant 6-17, 6-20
paramètres valeur 6-11, 6-19, 12-1
  constructeurs tableau ouvert 6-20
paramètres variable 6-19
paramètres variable (var) 6-11, 6-19, 12-1
parties variables (enregistrements) 5-24—5-26
partitionnement d'application 9-9
.pas, fichiers 2-3, 3-3, 3-8
pascal (convention d'appel) 6-5, 12-2
  constructeurs et destructeurs 12-4
  Self 12-4
PByteArray, type 5-30
PChar type 4-10
PChar, type 4-5, 5-14, 5-15, 5-16, 5-30, 5-48
  comparaison 4-13
PCurrency, type 5-30
périphériques, fonctions 8-5
PExtended, type 5-30
PGUID 10-3
pilotes de périphérique de fichiers texte 8-5
platform, directive 4-19
Pointer, type 5-27, 5-28, 5-29
pointeur
  types 5-29—5-30
pointeurs 5-27—5-30
  arithmétiques 4-10
  caractère 5-30
  chaînes à zéro terminal 5-15, 5-17
  chaînes longues 5-17
  constantes 5-48
  dans les paramètres tableau ouvert variant 6-17
  dans les paramètres var 6-13
  enregistrements et 5-25
  fichiers et 5-27
  fonctions 4-13, 5-31
  mémoire 11-5
  nil 5-29, 11-5
  opérateurs 4-10
  pointeurs de méthode 5-31
  présentation 5-28
  type Pointer 4-13, 11-5
  types pointeur 4-13, 5-28, 5-29—5-30, 11-5
  types procéduraux 4-13
  types procédure 5-30—5-33
  types standard 5-30
  variants et 5-33
pointeurs de méthode 4-13, 5-31
pointeurs de procédure 4-13
pointeurs procédure 5-31
POleVariant, type 5-30
polymorphisme 7-10, 7-12, 7-15
portée 4-30—4-32
  classes 7-3
  enregistrements 5-24
  identificateurs de type 5-40
Pred, fonction 5-3, 5-4
prédécesseur 5-3
prédéfinis, types 5-1
priorité des opérateurs 7-27
procédure
  constantes 5-48
  types 5-30—5-33, 11-11
procédures 3-4, 6-1—6-20
  appels de procédures 4-20, 6-1, 6-2, 6-19—6-20
  appels externes 6-7
  assembleur 13-16
  déclaration 6-2, 6-6
  imbriquées 5-32, 6-10
  pointeurs 4-13, 5-31

redéfinition 6-6, 6-8
procédures de sortie 9-6, 12-5—12-6
paquets et 12-5
procédures et fonctions surchargées
bibliothèques à chargement dynamique 9-6
program (mot réservé) 3-2
programme, contrôle 6-19
programmes 2-1, 3-1—3-9
contrôle 12-1—12-6
exemples 2-3—2-6
syntaxe 3-1—3-3
programmes exemple 2-3—2-6
projets 2-6, 3-7
propriété par défaut (objet COM) 5-36
propriétés 7-1, 7-18—7-25
comme paramètres 7-19
déclaration 7-18, 7-21
default 10-2
écriture seule 7-21
enregistrement 7-6
interfaces 10-4
lecture seule 7-21
redéfinition 7-7, 7-24
spécificateurs d'accès 7-19
tableau 7-6, 7-21
propriétés default
interfaces 10-2
propriétés par défaut 7-22
propriétés tableau
dans les interfaces de répartition 10-12
par défaut 7-22
spécificateurs de stockage et 7-24
prototypes 6-1
PShortString, type 5-30
PString, type 5-30
PTextBuf, type 5-30
Ptr, fonction 5-29
PVariant, type 5-30
PVarRec, type 5-30
PWideChar, type 5-14, 5-15, 5-30
PWideString, type 5-30
PWordArray, type 5-30

Q

qualifiés
identificateurs 3-7
QueryInterface, méthode 10-2, 10-5, 10-11
QWORD, type (assembleur) 13-14

R

raise (mot réservé) 4-22, 7-30, 7-32, 7-33
rang
types énumérés 5-7, 5-8
Read, procédure 8-3, 8-4, 8-6, 8-7
Readln, procédure 8-6, 8-7
readonly (directive) 10-2, 10-12
real (virgule flottante), opérateurs 4-7
Real, type 5-10
Real48, type 5-10, 7-6, 11-4
$REALCOMPATIBILITY, directive 5-10
ReallocMem, procédure 5-42, 11-1
redéfinition
méthodes 7-13
redéfinition des méthodes 7-11
masquer 7-12
redéfinition des propriétés 7-24
Automation 7-7
masquer 7-24
spécificateurs d'accès et 7-24
réels
types 11-4
réels, paramètres 6-19
réels, types 5-10
références circulaires
paquets 9-11
unités 3-8—3-9
références d'unité indirectes 3-7—3-8
références d'unité multiples 3-7—3-8
références de classe
types 7-25, 11-12
références mémoire (assembleur) 13-12
register (convention d'appel) 6-5, 7-7, 7-14, 7-16, 12-2
bibliothèques à chargement dynamique 9-4
constructeurs et destructeurs 12-4
interfaces 10-3, 10-7
Self 12-4
registres 6-5, 12-2, 12-3
assembleur 13-2, 13-9, 13-12, 13-17
stockage des ensembles 11-7
reintroduce (directive) 7-13
_Release, méthode 10-2, 10-5, 10-10
Rename, procédure 8-7
répartition des appels de méthodes 7-12
répartition des messages 7-18
requires, clause 9-9, 9-10, 9-11
.RES, fichiers 2-2
Reset, procédure 8-2, 8-4, 8-6, 8-7
resident (directive) 9-5
resourcestring (mot réservé) 5-45
Result, variable 6-3, 6-4
RET, instruction 13-3
retour à la ligne 4-5
retour chariot 4-1, 4-5
Rewrite, procédure 8-2, 8-4, 8-6, 8-7
Round, fonction 5-4
routines 6-1—6-20
Voir aussi fonctions, procédures
exportation 9-5
routines imbriquées 5-32, 6-10
routines standard 8-1—8-11
chaînes à zéro terminal 8-7, 8-8
chaînes de caractères étendus 8-8
RTTI 7-5, 7-14, 7-23
$RUNONLY, directive 9-13

S

$S, directive 11-2
safecall (convention d'appel) 6-5, 12-2
constructeurs et destructeurs 12-4
interfaces 10-3
interfaces doubles 10-14
Self 12-4
sans type
fichiers 8-4
sans type, fichiers 5-27
sans type, paramètres 6-14
scalaire 5-3
scalaires, types 5-3—5-9
Seek, procédure 8-3
SeekEof, fonction 8-6
SeekEoln, fonction 8-6
sélecteur (enregistrements) 5-25

Self 7-10
conventions d'appel 12-4
méthodes de classe 7-28
sémantique
copie-par-écriture 5-13
séparateurs 4-1, 4-6
set *Voir* spécificateur write
SetLength, procédure 5-11, 5-17, 5-20, 5-22, 6-16
SetMemoryManager, procédure 11-2
SetString, procédure 5-17
ShareMem, unité 9-8
shl 4-9
Shortint, type 5-4, 11-3
ShortString, type 5-11, 5-12, 5-30, 11-6
paramètres 6-14
tableaux variant et 5-37
ShowException, procédure 7-35
shr 4-9
signe
dans les transtypages 4-16
nombres 4-4
simples, types 5-3
Single, type 5-10, 11-4
SizeOf, fonction 5-2, 5-5, 6-16
Smallint, type 5-4, 11-3
.so, fichiers 9-1
soulignés 4-2
sous-intervalle, types 4-8
soustraction 4-7
pointeurs 4-10
spécificateur de stockage 7-23
spécificateur default 7-7, 7-19, 7-23
spécificateur index 7-7, 7-18, 7-22
spécificateur nodefault 7-7, 7-19, 7-23
spécificateur read 7-7, 7-19
interfaces objet 10-2, 10-4, 10-7
propriétés tableau 7-21
redéfinition 7-14, 7-19
spécificateur index 7-22
spécificateur stored 7-7, 7-19, 7-23
spécificateur write 7-7, 7-19
interfaces objet 10-2, 10-4
propriétés tableau 7-21
redéfinition 7-14, 7-19
spécificateur index 7-22
spécificateurs d'accès 7-1, 7-19
Automation 7-7
convention d'appel 6-5, 7-19
propriétés tableau 7-21
redéfinition 7-14, 7-19, 7-24
spécificateurs index 7-23
spécificateurs de stockage
propriétés tableau et 7-24
standard
routines 8-1—8-11
statiques
méthodes 7-11
tableaux 5-19, 11-8
stdcall (convention d'appel) 6-5, 12-2
bibliothèques partagées 9-4
constructeurs et destructeurs 12-4
interfaces 10-3
Self 12-4
Str, procédure 8-7
StrAlloc, fonction 5-42
StrDispose, procédure 5-42
string (mot réservé) 5-11
StringToWideChar, fonction 8-8
StrToInt64, fonction 5-4
StrToInt64Def, fonction 5-4
structures 5-23
structurés
types 5-18
StrUpper, fonction 5-16
Succ, fonction 5-3, 5-4
successeur 5-3
surcharge des implémentations d'interface 10-6
surcharge des méthodes 10-6
symboles 4-1, 4-2
assembleur 13-10
symboles spéciaux 4-1, 4-2
symboles, paires 4-2
syntaxe
descriptions 1-2
formelle A-1—A-6
syntaxe étendue 4-5, 5-15, 6-1, 6-4
System, unité 3-6, 5-30, 5-34, 5-36, 6-17, 7-3, 8-1, 8-8, 10-2, 10-3, 10-5, 10-11, 11-12
bibliothèques à chargement dynamique 9-6, 9-7
clause uses et 8-1
gestion de la mémoire 11-2
modification 8-1
portée 4-32
SysUtils, unité 3-6, 5-30, 6-10, 6-17, 7-28, 7-29, 7-31, 7-35
bibliothèques à chargement dynamique 9-8
clause uses et 8-1

T

$T, directive 4-13
tableau
propriétés 7-21
tableau ouvert variant
paramètres
tableaux
array of const 6-17
tableau, propriétés 7-6
tableaux 5-3, 5-19—5-23
accès avec PByteArray 5-30
accès avec PWordArray 5-30
affectations et 5-23
caractère 4-5, 5-20
caractères 5-14, 5-16, 5-18
constante 5-46
constantes chaîne et tableaux de caractères 5-46
constructeurs tableau ouvert 6-17, 6-20
dynamiques 5-43
index 4-16
multidimensionnels 5-20, 5-22
paramètres 6-11, 6-15
statiques 5-19
tableaux de caractères et constantes chaînes 5-15
variants et 5-33, 5-34, 5-36
tableaux compactés 4-5, 4-10, 5-20
tableaux dynamiques 5-20, 11-8
affectation aux 5-21
comparaison 5-21
dans les bibliothèques à chargement dynamique 9-8
enregistrements et 5-25
fichiers et 5-27
gestion de la mémoire 11-3
libération 5-43
multidimensionnels 5-22
paramètres tableau ouvert et 6-15
tronquer 5-21
variants et 5-34
tableaux multidimensionnels 5-47
tableaux statiques 11-8
variants et 5-33
tableaux variant 5-34
chaînes 5-37

tables des méthodes
virtuelles 11-11
TAggregatedObject 10-7
taille de pile 11-2
tas 11-2
tas, mémoire 5-42
TBYTE, type (assembleur) 13-14
TClass 7-3, 7-25, 7-27
TCollection 7-26
Add, méthode 7-10
TCollectionItem 7-26
TDateTime 5-36
Text, type 5-26
TextBuf, type 5-30
texte
fichiers 8-2, 8-3
texte, type 8-3
TextFile, type 5-26
TGUID 10-3
then (mot réservé) 4-24
thread, variables 5-43
threadvar 5-43
TInterfacedObject 10-2, 10-5
to (mot réservé) 4-29
TObject 7-3, 7-18, 7-27
tokens 4-1
TPersistent 7-6
transtypages 4-16—4-18, 7-8
avec vérification 7-27
dans les déclarations de
constantes 5-44
interface 10-10
paramètres sans type 6-14
variants 5-34
vérification 10-11
transtypages de valeur 4-16
transtypages de variable 4-16,
4-17
True 5-6, 11-3
Trunc, fonction 5-4
try...except, instructions 7-30,
7-31
try...finally, instructions 7-34
TTextRec, type 5-30
Turbo Assembler 13-1, 13-5
TVarData 5-34
TVarRec 5-30
TVarRec, type 6-17
TWordArray 5-30
type de la valeur renvoyée
(fonctions) 6-3
types 5-1—5-41
assembleur 13-13
booléens 5-6, 11-3
caractères 5-5, 11-3
chaînes 5-11—5-17, 11-6, 11-7
classe 7-2, 7-7, 7-8, 7-9, 7-18,
11-11
classification 5-1
compatibilité 5-18, 5-31,
5-38, 5-39
compatibilité pour
l'affectation 5-39
constantes 5-44
déclaration 5-40
déclarés 5-1
définis par l'utilisateur 5-1
enregistrement 5-23—5-26,
11-8
ensemble 11-7
énumérés 5-6
exception 7-29
fichier 11-9
fondamentaux 5-2
formats de données
internes 11-3—11-13
génériques 5-2
identité 5-38
intégrés 5-1
interface 10-1—10-4
intervalle 5-8
objet 7-4
portée 5-40
prédéfinis 5-1
réels 5-10
simples 5-3
tableau 5-19—5-23, 11-8
variant 5-33—5-37
types automatisables 7-6, 10-12
types de base 5-8, 5-18, 5-19,
5-20
types de données *Voir* types
types de référence de classe 7-25
constructeurs et 7-26
types entier
comparaison 4-12
conversion 4-17
types entiers 5-4
constantes 5-44
formats de données 11-3
types énumérés 11-4
publier 7-6
valeurs anonymes 5-8, 7-6
types interface de
répartition 10-11
types objet 7-4
types Pointer 11-5
types procéduraux 4-17
types procédure 5-30—5-33
appel de bibliothèques à
chargement dynamique 9-2
appel des routines avec 5-32,
5-33
compatibilité 5-31
dans les affectations 5-32,
5-33
mémoire 11-11
paramètres par défaut 6-18
types réels 11-4
comparaison 4-12
conversion 4-17
publication 7-6
types référence de classe
comparaison 4-13
mémoire 11-12
variants et 5-33
types scalaires 5-3—5-9
types sous-intervalle 4-26
types structurés 5-18
enregistrements et 5-25
fichiers et 5-27
variants et 5-33

U

UCS-2 5-14
UCS-4 5-14
Unassigned (variants) 5-34, 5-36
Unicode 5-5, 5-13, 5-14
union (ensembles) 4-11
UniqueString, procédure 5-17
unités 2-1, 3-1—3-9
portée 4-32
syntaxe 3-3—3-9, 4-19
unités mutuellement
dépendantes 3-8
until (mot réservé) 4-28
UpCase, fonction 5-12
uses, clause 2-1, 3-1, 3-3, 3-4,
3-5, 3-5—3-9
interface, section 3-8
ShareMem 9-9
syntaxe 3-6
unité System et 8-1
unité SysUtils et 8-1

V

Val, procédure 8-7
valeur
paramètres 12-1
valeur renvoyée (fonctions) 6-3,
6-4
constructeurs 7-14

valeurs anonymes (types énumérés) 5-8, 7-6
valeurs immédiates (assembleur) 13-12
var (mot réservé) 5-41, 6-11, 12-1
varargs (directive) 6-7
VarArrayCreate, fonction 5-36
VarArrayDimCount, fonction 5-37
VarArrayHighBound, fonction 5-37
VarArrayLock, fonction 5-37, 10-13
VarArrayLowBound, fonction 5-37
VarArrayOf, fonction 5-36
VarArrayRedim, fonction 5-37
VarArrayRef, fonction 5-37
VarArrayUnlock, procédure 5-37, 10-13
VarAsType, fonction 5-34
VarCast, procédure 5-34
variable (var)
  paramètres 12-1
variables 4-6, 5-41—5-43
  adresses absolues 5-42
  allouées sur le tas 5-42
  déclaration 5-41
  depuis les bibliothèques à chargement dynamique 9-1
  gestion de la mémoire 11-2
  globales 10-10
  initialisation 5-41, 5-42
  thread 5-43
variables de thread 5-43
variables dynamiques 5-42
  dans les bibliothèques à chargement dynamique 9-8
  et constantes pointeur 5-48
variables fichier 8-2
variables globales 5-41
  bibliothèques à chargement dynamique 9-7
  gestion de la mémoire 11-2
variables locales 5-41, 6-10
  gestion de la mémoire 11-2
variables thread
  dans les paquets 9-10
variants 5-33—5-37, 11-13
  chaînes et tableaux variant 5-37
  conversions 5-34, 5-34—5-36
  enregistrements et 5-25
  et Automation 11-13
  évaluation complète 4-9
  évaluation optimisée 4-9
  fichiers et 5-27
  fonctions non Delphi 11-13
  gestion de la mémoire 11-3, 11-12
  initialisation 5-34, 5-42
  interfaces et 10-10
  libération 5-43
  OleVariant 5-37
  opérateurs 4-7, 5-36
  tableaux variant 5-36
  transtypages 5-34
  type variant 5-30, 5-33—5-37
varOleString, constante 5-37
varString, constante 5-37
VarType, fonction 5-34
varTypeMask, constante 5-34
VCL 1-2
vérification
  transtypages
    objets 7-27
vérification de type (objets) 7-27
vérification des limites de compilation 5-5, 5-9
vérification, transtypages
  interfaces 10-11
vides
  ensembles 5-19
VirtualAlloc, fonction 11-1
VirtualFree, fonction 11-1
virtuelles
  méthodes 7-11
visibilité (membres de classe) 7-4
  interfaces 10-2
VMT 11-11
vraies, constantes 5-44

W

$WARNINGS, directive 4-19
$WEAKPACKAGEUNIT, directive 9-13
WideChar, type 4-10, 5-5, 5-12, 5-14, 5-30, 11-3
WideCharLenToString, fonction 8-8
WideCharToString, fonction 8-8
WideString, type 5-11, 5-13, 5-14, 5-30
  gestion de la mémoire 11-7
Windows 7-18
  et variants 11-12
  gestion de la mémoire 11-1, 11-2
  messages 7-16
Windows, unité 9-2
with, instructions 5-24
Word, type 5-4, 11-3
  assembleur 13-14
WordBool, type 5-6, 11-3
Write, procédure 8-3, 8-4, 8-6, 8-7
write, procédures 5-3
Writeln, procédure 2-4, 8-6, 8-7
writeonly (directive) 10-2, 10-12

X

$X, directive 4-5, 5-15, 6-1, 6-4
.xfm, fichiers 2-2, 2-8, 7-6
.xof, fichiers 2-3
xor 4-8, 4-9

Z

-$Z-, option de compilation 9-13
zéro terminal
  chaînes 5-14—5-17, 5-30