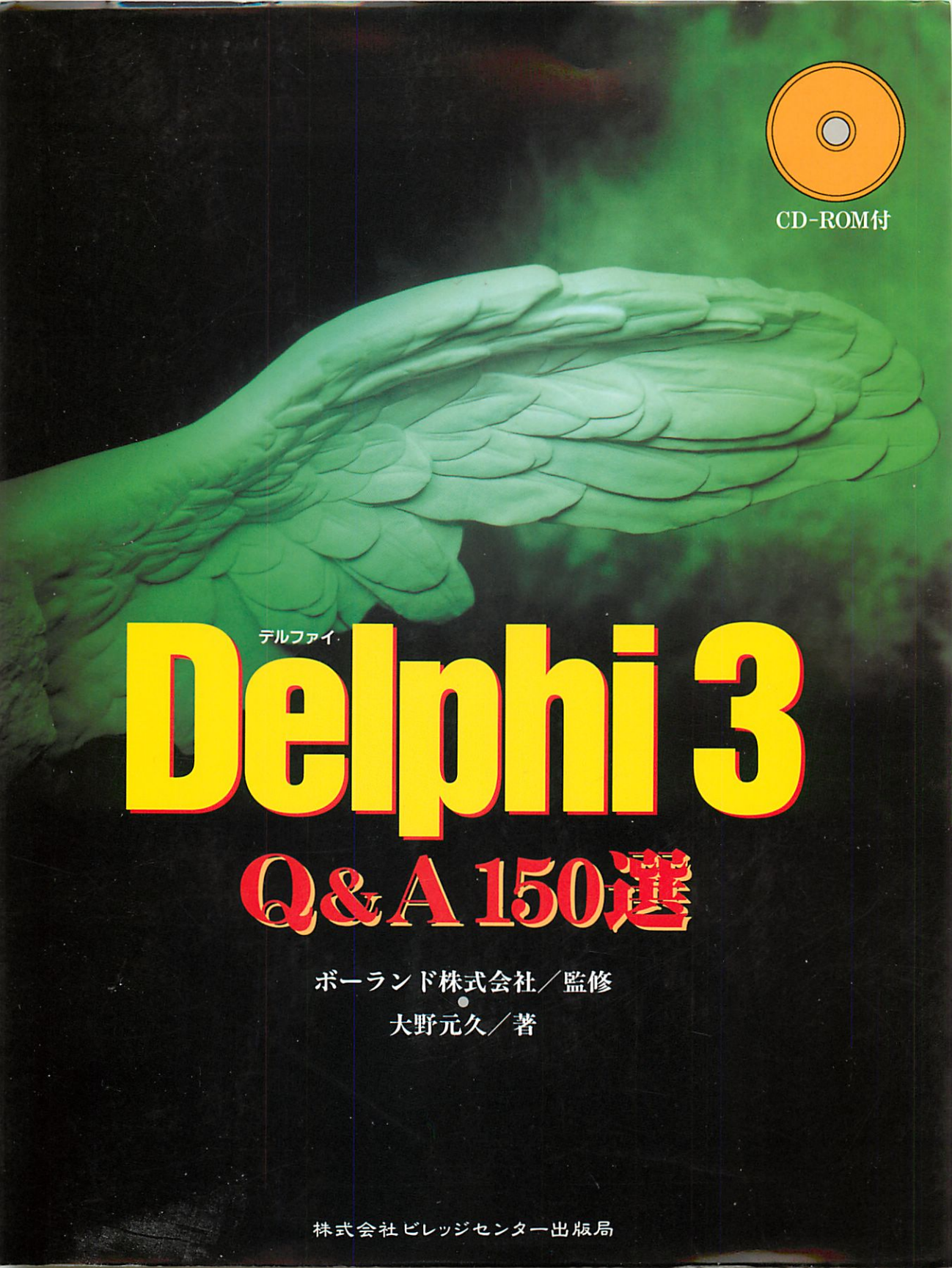


デルファイ Delphi 3

Q&A 150選

ボーランド株式会社／監修
大野元久／著

株式会社ビレッジセンター出版局

「Delphi 3 Q&A 150選」

Delphiの華々しい登場から2年あまりが過ぎ、ビジュアル開発ツールとして広く知られるようになりました。本書の前身である「Delphi Q&A 120選」「Delphi 2.0 Q&A 120選」は、この斬新な開発ツールを使いこなすための情報を提供してきました。今回、Delphi 3に対応するにあたり、項目の見直しや新しい項目を追加する以外に、少し方針を変更しました。

まず、イラストではなく画面を掲載することで、いくつかの項目については実際の使い方を確認できるようにしました。また、動作原理の理解よりも実際の問題解決を中心にすることを考えました。このため、いくつかの問題については、プログラムコードを提示する代わりに付録CDでコンポーネントを提供しています。

たとえば、Delphiで作成したアプリケーションでは、メインフォームのアニメーション動作（最小化や復元で、段々大きさが変化する動作）が使われません。これを実現するためには、かなり高度なプログラミングが必要ですが、本書の回答は独自のコンポーネントを配置するだけというものです。コンポーネントは、内部でどんな複雑な処理をしていても、利用者はその内部に立ち入ることなく簡単に使えます。これは、Delphi自身の特長を活かすことでもあります。

フォームのアニメーション以外にも、25個以上のコンポーネントがソースコード付きで収録されています。これらのコンポーネントやサンプルプログラムをお使いいただければ、あらためてDelphiの奥深さを感じていただけると思います。また、技術的な仕組みに興味のある方は、CD-ROMに収録されているソースコードを直接参照してください。

なお、本書の動作確認はDelphi 3 Client/Server SuiteおよびWindows 95によって行なっています。本書によって、Delphiへの理解が進み、さまざまな場で活用されることを願ってやみません。

最後になりますが、編集を担当されたビレッジセンターの多田尚弘氏には、いつもながら短時間での作業を強いることになりました。また、読者の方からのお便りにも励まされることも多くありました。この場を借りて、氏ならびに読者の方々に心より感謝いたします。

ボーランド株式会社　マーケティング部　大野　元久

「Delphi 3 Q&A 150選」利用法

本書には、Delphi 3に関して特にパソコン通信で寄せられた質問に対する回答を記載しています。また、多くの質問について単なる可能、不可能だけでなく具体例などを交えて説明してあります。

ここに記載しているほとんどの内容はDelphi 3に含まれるドキュメントやVCLソースコード、Windows APIなどに公開されている情報を元に作成したものですが、ボーランドのテクニカルサポートでは本書に関する質問は受けません。誤りや、よりよい解決策などの指摘などはビレッジセンター宛にご連絡ください。原則としてテクニカルサポートセンターでは個々の技術的な質問については受け付けていませんが、本書ではそうした質問に対してもできるだけ具体的な解決策を提供しようとしています。

ただし、マニュアルやオンラインヘルプ、添付のドキュメントを読めば簡単にわかるようなものはあまり掲載していません。その意味で、本書の内容はやや高度なトピックを含んでいるものもあります。なお、マニュアル、ドキュメントファイル、各種のサンプルプログラムは、Delphiを理解する上で非常に役立つものとなるでしょう。是非時間をかけて通読されることをお勧めします。

本書に記載されているプログラムリストの多くは、フォーム上に配置したボタン(Button)のイベントとして定義されています。フォーム上にボタンを配置し、イベントハンドラにプログラム例の内容を定義することは、そのプログラムの動作を確認する簡単な手段です。さらに、結果を表示したい場合はListBoxやMemoコンポーネントに文字列を追加したり、ShowMessageで表示しています。

そして、こうしたプログラムではリストの一部を省略しているものがあります。たとえば、フォームユニットにおいて型を宣言するインターフェース部(interface)と実装部(implementation)の間には、必ずimplementationという予約語や{$R *.DFM}というリソース指令が必要ですが、ほとんどのプログラムではこれらを記載していません。完全なプログラムリストは、添付のディスクに含まれているファイルを参照してください。なお、コンポーネントを作成するプログラムについては全プログラムリストを掲載しています。

本書は現在発売されているWindows95における日本語Delphi 3について言及しています。多くの内容はWindows NTや英語版Delphi、将来のバージョンでも応用できると思いますが、動作については確認していません。一部の手法はドキュメントされた内容ではなく、VCLソースコードに依存しているものもあります。こうした手法は、将来のバージョンでは使えなくなる可能性もあります。SQLサーバーに関するプログラミングについては本書では取り上げていません。

本書の目的は、単なるQ&Aにとどまりません。本書は、Delphiの仕組みや特長について理解を深めることにあります。このため、ひとつの質問に対して解説が長いものや、やや難しいトピックを含んでいるものもあります。しかし、トラブルシューティングの目的以外でも本書を読んでいただければ、今まで気づかなかったようなDelphiの機能が少なからず発見できるでしょう。

表記規則

本文中では、やさしい方から

 Q&Aが容易に理解できることを表しています。

 Q&Aを理解するために、ある程度の知識を必要とすることを表しています。

 Q&Aを理解するために、かなりの知識を必要とすることを表しています。

 Q&Aを理解するために、非常に高度な知識を必要とすることを表しています。

として記載してあります。

難易度マークは、あくまでめやすとしてご利用ください。質問に対する答えが付録CDのコンポーネントで容易に解決する場合でも、その内部構造を理解するためには高度な知識を必要とすることがあります。

その他の記号の意味は、下記のとおりです。

 オンラインヘルプを参照することを意味しています。

 注意すべきことや、特に重要な点についての説明です。

 付録ＣＤに収録されているサンプルプロジェクト名です。

プログラムファイルに関して

質問に対する回答で、付録CDに関連するファイルやプログラムが収録されているものは、「付録CD」としてプロジェクト名(.DPR)が記載されています。プロジェクトに関するすべてのファイルについては、付録CDのREADME.TXTを参照してください。また、コンポーネントや特別なユニットについては、ユニットファイル名が記載されています。なお、複数の回答で同じプロジェクトを参照している場合もあります。

サンプルプログラムの中には、付録CDで提供されるコンポーネントを使っているものもあります。使用するコンポーネントについても記載してありますが、あらかじめQACOMPOディレクトリのコンポーネントをDelphi 3に登録しておくことをお勧めします。

CONTENTS

第1章　統合開発環境

第2章　アプリケーション／フォーム

CONTENTS

第3章　プログラミング

CONTENTS

第4章　コンポーネント

CONTENTS

第5章　データベース

第6章　for Visual Basicプログラマ

第7章　for C/C++プログラマ

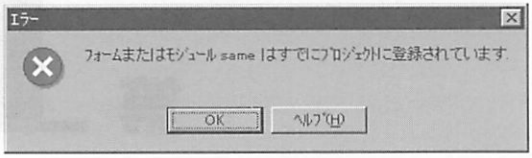
第1章

統合開発環境

本章では、Delphiの統合開発環境の使い方を含むプログラミング以外の質問を取り上げています。

Q. プロジェクトを保存するときにフォームとプロジェクトに同じ名前を付けようとすると「フォームまたはモジュールXXXは、すでにプロジェクトに登録されています」というエラーが発生します（図1-1）。同じ名前では保存できないのでしょうか。

図1-1　プロジェクト名とフォーム名の重複エラー



Delphiでは、フォームとプロジェクトに同じ名前を付けることはできません。

Delphiのフォームは拡張子が.DFMというフォームファイルと.PASというユニットファイルとで成り立っています。

.DFMファイルは、フォームやフォーム上に配置したコンポーネントのプロパティやイベントハンドラなどの名前を保持するバイナリファイルです。このファイルは、ビジュアルに設計した内容を保持するもので、通常は他のテキストエディタなどでは扱えません。

.PASファイルは、フォームファイルに対応するイベントハンドラなどの手続きや関数を記述したプログラムファイルです。.PASファイルの先頭には「**unit** ユニット名」という記述があり、Object Pascalのユニットであることを示しています。

もともと、Object Pascalの元になっているボーランドのTurbo Pascalでは、プログラムを複数のソースコードに分けて開発するためにユニットという手法を使っていました。メインプログラムは、「**program** プログラム名」という記述ではじまります。それ以外のプログラムはユニットとなります。通常、プログラム名やユニット名は保存するファイル名と同じ名前を付けるため、それぞれ異なる名前にしておく必要があります。

Delphiでは、すべてのフォームに対するユニットファイルをまとめているのが、プロジェクトソースというプログラムファイルです。メインメニューで**[表示(V)｜プロジェクトソース(J)]**を選ぶと、プロジェクト全体を管理するためのメインプログラムが表示されます。このプログラムの拡張子は.DPRとして保存されていますが、実際には他のユニットと同じくObject Pascalのプログラムです。拡張子が違うのは、他の一般的なユニットファイルと区別するためです。

以上の理由により、プログラム名とユニット名に同じ名前を付けることはできません。

また、フォームファイル名がそのままユニット名として使われるため、保存するファイル名とフォームの名前を同じにすることはできません。これは、いずれもグローバルな識別子になるため同じ名前を許すと両者を区別できなくなるためです。

Q. Delphiでプロジェクトを新規に作成し、保存するときに必ずDelphiのディレクトリがデフォルトになっています。[プロジェクト(P)｜オプション(O)｜ディレクトリ/条件]で[出力ディレクトリ(O)]を指定しても、変化はありません。デフォルトのディレクトリを指定することはできないのでしょうか。

[Borland Delphi 3]グループのDelphi 3ショートカットで右クリックし、[プロパティ(R)]を選びます。あるいは、Windows 95の[スタート]、[設定(S)]、[タスクバー(T)]で[タスクバーのプロパティ]ダイアログを表示させ、[スタートメニューの設定]ページで[詳細]ボタンを押し、[プログラム｜Borland Delphi 3｜Delphi 3]で右クリックして、[プロパティ(R)]を選びます。

ここで[ショートカット]ページの[作業フォルダ(S)]を変更すれば、Delphiがファイルを保存するときのデフォルトのディレクトリを変更できます（図1-2）。通常、このディレクトリはDelphiをインストールしたディレクトリ（C:¥Program Files¥Borland¥Delphi 3など）になっています。つまり、特に何も変更しなければファイルを保存するときにカレントディレクトリがこのディレクトリになります。

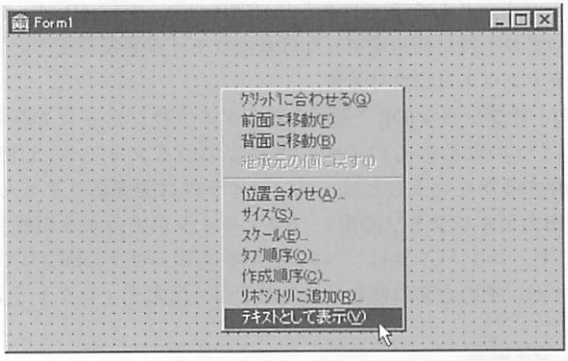
[プロジェクト(P)｜オプション(O)]で表示されるダイアログボックスのすべてのページの項目は、コンパイラのためのものです。作成しているフォーム(.DFM)やユニット(.PAS)をどこに保存するかということは、コンパイラには関係ありません。[ディレクトリ/条件]ページの[出力ディレクトリ(O)]は、「コンパイラが出力する場所」を指定するという意味になります。コンパイラは、ユニットオブジェクト(.DCU)や実行ファイル(.EXE)を出力しますが、そのときに使われるのがこの「出力ディレクトリ」です。

図1-2　作業フォルダの変更

Q. 設計したフォームに余計なコンポーネントが追加されていたり、誤ってプロパティやイベントハンドラを設定していないかどうかを調べるために、フォームの情報をテキスト形式で表示することはできないでしょうか。

ビジュアル開発の場合は、すべてをソースコードで記述する場合と違ってプログラムを1行ずつ追いかけることができません。Delphi 3では、フォームのスピードメニューで[テキストとして表示(V)]を選べば、フォームの情報がコードエディタにテキスト形式で表示されます（図1-3）。このとき、コードエディタにはユニットソースコード（.PAS）は表示されません。また、そのフォームから継承しているフォームがあれば、あらかじめ継承したフォームもテキスト表示に切り替える必要があります。

図1-3　フォーム情報をテキストで表示する



フォームは、次のようなテキストに展開されます。これを編集した後でスピードメニューの[フォームとして表示(V)]を選ぶと、再びビジュアルなフォームとして表示されます。

```
object Form1: TForm1
  Left = 200
  Top = 107
  Width = 435
  Height = 300
  Caption = 'Form1'
  Font.Charset = SHIFTJIS_CHARSET
  Font.Color = clWindowText
  Font.Height = -12
  Font.Name = 'ＭＳ Ｐゴシック'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 12
end
```

もし、フォームそのものではなく特定のコンポーネント、または配置したすべてのコンポーネントの情報をテキストで確認したいのであれば、フォーム上で必要なコンポーネントをまとめて選択して**[編集(E)｜コピー(C)]**を選び、コードエディタや他のテキストエディタに切り替えてから**[編集(E)｜貼り付け(P)]**を選びます。Delphiの2Way-Toolという機能によって、ビジュアル開発の内容は自動的にテキスト形式に変換されます。

また、DOSプロンプトではCONVERT.EXEというコマンドラインユーティリティを使って、フォームファイル(.DFM)とテキストファイル(.TXT)を相互に変換できます。ただし、フォーム上にビットイメージやアイコンがある場合は、イメージデータがすべて16進データとしてテキストに変換されるため、膨大な大きさのファイルになることがあります。この場合は、いったんイメージやアイコンを初期化しておく方がよいでしょう。

パネルのような、他のコンポーネントを配置できるコンテナコンポーネントを使っている場合は、コンテナコンポーネントの定義中に表示されます。本来、コンテナコンポーネント上にあるべきものが、この中になければコンテナ上に置かれていないということが考えられます。

```
object AboutBox: TAboutBox
  Left = 200
  Top = 108
  BorderStyle = bsDialog
  Caption = 'ﾊﾞｰｼﾞｮﾝ情報'
  ClientHeight = 213
  ClientWidth = 298
  Font.Charset = SHIFTJIS_CHARSET
  Font.Color = clWindowText
  Font.Height = -12
  Font.Name = 'ＭＳ Ｐゴシック'
  Font.Style = []
  Position = poScreenCenter
  PixelsPerInch = 96
  TextHeight = 12
  object Panel1: TPanel
    Left = 8
    Top = 8
    Width = 281
    Height = 161
    BevelInner = bvRaised
    BevelOuter = bvLowered
    TabOrder = 0
    ...
```

フォームの継承を使っている場合は、先頭の行は**object**ではなく**inherited**から始まります。また、継承したオブジェクトでプロパティが変更されているものは、**inherited**～**end**の間に変更されたプロパティだけが表示されます。

Q. フォームを設計した後、誤ってコンポーネントを移動しないように位置を固定しておくことはできませんか。

すべてのコンポーネントを配置したフォームで、後はオブジェクトインスペクタでプロパティやイベントハンドラだけを設定するという場合でも、コントロールを選択するためにクリックしようとして誤ってドラッグして移動させてしまうかもしれません。

このような場合は、[編集(E)｜コントロールのロック(K)]を選んでおくと、すべてのコントロール（ビジュアルコンポーネント）を移動できないように固定しておくことができます。この場合、すべてのコンポーネントの位置が固定されますが、タブ順序や前後関係(Zオーダー）は変更できます。また、新しくコンポーネントを配置することはできますが、いったん配置した位置を変更することはできません。ロックを解除するためには、もう一度[編集(E)｜コントロールのロック(K)]を選びます。

Q. フォームにパネルを配置してAlignプロパティをalClientにしているため、フォームをクリックする場所が隠れています。フォームのプロパティを変更するためには、どうすればよいでしょうか。また、フォーム上のコンポーネントをドラッグの範囲指定で選択するように、パネル上のコンポーネントを選択することはできませんか。

フォームやフォームに配置したコンポーネントは、オブジェクトインスペクタのタイトルバーのすぐ下にあるオブジェクトセレクタという場所で選択できます。

フォーム上で一つのコンポーネントを選択している場合は、そのコンポーネントを〔Shift〕を押しながら左クリックで選択することでフォーム自身を選ぶことができます。〔Shift〕+左クリックは、複数のコンポーネントの選択・非選択状態を切り替えるものです。

また、他のコンポーネントを配置するコンテナとして使われているパネルなどのコンポーネントは、フォーム上のように単純にマウスのドラッグによる範囲指定をしようとしても、パネルそのものを移動することになってしまいます。この場合は、〔Ctrl〕を押しながらドラッグすれば、ドラッグした範囲のコンポーネントをまとめて選択できます。

Q. アプリケーションをコンパイルしたり構文チェックをするときに、コンパイル行数などがまったく表示されません。Borland C++やTurbo C++のように進行状況を表示することはできないでしょうか。

通常は、少しでもコンパイル時間を短縮するためにコンパイル状況を表示していません。[ツール(T)｜環境オプション(O)｜設定]で[コンパイル状況の表示(C)]をチェックすれば、コンパイル行数が表示されるようになります。

Q. オブジェクトインスペクタの[イベント]ページで、イベントハンドラを定義しようとしてダブルクリックしたのですが、誤って別のイベントを選んでしまいました。簡単に定義を削除することはできませんか。

イベントの項目をダブルクリックしてプロトタイプが生成されても、プログラムをコンパイルするときや実行するときまでに何も入力しなければ、そのプロトタイプは自動的に削除されます。もし、空のイベントハンドラだけでも残しておきたい場合には空のコメント（{}）などを記述しておいてください。

Q. コンポーネントをまとめて選択して、もっとも左に寄せたいのですが、フォームのスピードメニューの[位置合わせ(A)]で[左寄せ(N)]を選んでも別の場所に調整されてしまうことがあります。

Delphiの位置合わせ機能は、最初に選択したコンポーネントを基準にします。つまり、すべてのコンポーネントを一番左側に合わせるのであれば、最初にもっとも左側のコンポーネントを選ぶ必要があります。右寄せや上に寄せる場合も同様です。ただし、等間隔に配置する場合は、配置されている位置の順で並び換えられます。

Q. ごく単純なプログラムを作成しても、実行ファイルの大きさが180KB程度になってしまいます。実行ファイルの大きさを小さくする方法はないでしょうか。

Delphiは、単独で実行できるアプリケーションを作成できるのが一つの特長ですが、これはDelphiが使うすべての機能が実行ファイルの中に埋め込まれるということも意味します。つまり、フォームやコンポーネントを処理するプログラムコードも実行ファイルの中に埋め込まれます。

しかし、Delphi 3ではパッケージという機能が採用され、フォームやコンポーネントに関する処理とアプリケーション側のプログラムコードを分割できるようになりました。パッケージを使うためには、[プロジェクト(P)｜オプション(O)]で、[実行時パッケージを使って構築(B)]チェックボックスをチェックしてください。実行ファイルにフォームやコンポーネントのためのコードが埋め込まれなくなるため、サイズを大幅に小さくできます(図1-4)。

図1-4　パッケージを使う設定

パッケージを使って構築されたアプリケーションを実行するには、BINディレクトリ(または¥Windows¥Systemディレクトリ)にある*.DPLという実行時パッケージが必要になります。これは、最低でも1.5MB以上の大きさになりますので、配布するアプリケーションの数が少ない場合は、パッケージを使わない方が配布サイズが小さくなることもあります。

なお、Delphi 3で配布できるパッケージファイルの種類については、Delphi 3ディレク

トリにあるDEPLOY.TXTに書かれています。パッケージの詳細については、『言語ガイド』の第15章やオンラインヘルプを参照してください。

また、Delphiでは、ビジュアルプログラミングを使わずにWindows APIだけでプログラムすることができます。たとえば、**[ファイル(F)｜アプリケーションの新規作成(T)]**で新しいプロジェクトを作成し、**[表示(V)｜プロジェクトマネージャ(P)]**を使ってプロジェクトマネージャからユニットUnit1を削除すると、空のプロジェクトができあがります。プロジェクトソースは、**[表示(V)｜プロジェクトソース(J)]**で表示できます。ここに以下のようなプログラムを入力すれば、非常に小さいアプリケーションを作成できます。

```
program SmallApp;

uses Windows;

{$R *.RES}

begin
  MessageBox(0, 'Hello, Delphi Programmer!', 'SmallApp', MB_OK);
end.
```

CHAP1¥SMALLAPP.DPR

Q. **コンポーネントを作成して、[コンポーネント(C)｜インストール(I)]でインストールしたのですが、コンポーネントパレットに表示させるアイコンを指定する方法がわかりません。**

まず、**[ツール(T)｜イメージエディタ]**でコンポーネントファイル名に対応するリソース(.DCR)を作成します。このリソースに新たにビットマップリソース(16×16～20×20程度)を新規に作成し、リソース名をコンポーネントのクラス名(TTEXTなど)と同じにします。ここで、リソース名はすべて大文字にします。

このファイルをコンポーネントのソースコードと同じディレクトリに置いておきます。こうしておけば、自動的にこのビットマップがコンポーネントパレットに登録されるアイコンとして使われます。コンポーネントリソースが作成されていないか、該当するクラス名のビットマップリソースが見つからない場合は、上位クラスのビットマップがそのまま使われます。

Q. コードエディタで、同じキー操作を繰返し実行するための機能はありませんか。

〔Ctrl〕+〔Shift〕+〔R〕を押すと、以後入力するキー操作を記録が始まり、ふたたび〔Ctrl〕+〔Shift〕+〔R〕を押すまで続けて記録されます。〔Ctrl〕+〔Shift〕+〔P〕を押すと、記録されたキー操作が自動的に実行されます。

たとえば、挿入状態で行頭にカーソルを移動させ、〔Ctrl〕+〔Shift〕+〔R〕、〔/〕、〔/〕、〔Ctrl〕+〔←〕、〔↓〕、〔Ctrl〕+〔Shift〕+〔R〕と入力すると、以後は〔Ctrl〕+〔Shift〕+〔P〕を入力するだけで連続する行をコメントにできます。

この他、〔Ctrl〕+〔K〕に続いて数字（〔0〕～〔9〕）を入力することで10個のカーソル位置を記憶することができます。記憶した位置へジャンプするには〔Ctrl〕+〔Q〕に続いて対応する数字を入力します。記憶した位置はしおりとして、エディタの左端にアイコンが表示されます。しおりは、プログラムを編集した後でも正しい位置を表示できるよう行の追加・削除によって自動的に移動します。デフォルトのキーマップを使っている場合、〔Ctrl〕+〔2〕のようなキー操作で直接しおりの場所に移動できます。

エディタのキー操作については、オンラインヘルプの[Delphiの使い方｜プログラミング環境｜キーボードショートカット]を参照してください。

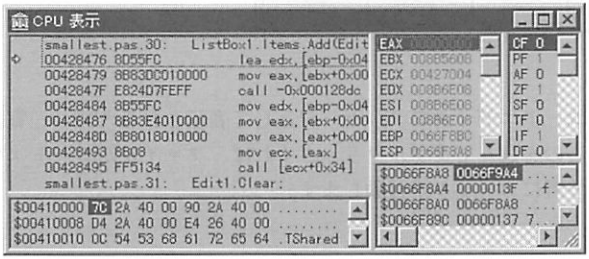
Q. アセンブリレベルでのデバッグはサポートされていませんか。

正式に公開された機能ではありませんが、レジストリを設定することでCPUウィンドウを表示できるようになります。具体的には、HKEY_CURRENT_USER¥Software¥Borland¥Delphi¥3.0¥DebuggingキーでEnableCPUを1に設定します。また、次のプログラムを実行しても設定できます。

```
uses Registry;

procedure TForm1.Button1Click(Sender: TObject);
var
  Reg: TRegIniFile;
begin
  Reg := TRegIniFile.Create('Software¥Borland¥Delphi¥3.0');
  Reg.WriteInteger('Debugging', 'EnableCPU', 1);
  Reg.Free;
end;
```

このレジストリを設定した後、Delphi 3を起動すると[表示(V)]メニューに[CPUウィンドウ(D)]という項目が追加されていることがわかります。これを使うと、図1-5のようなCPUウィンドウが表示され、逆アセンブルリストやレジスタ、データなどが表示できます。

図1-5　CPUウィンドウ



逆アセンブルビューは公式な機能ではないため、動作についての保証はありません。

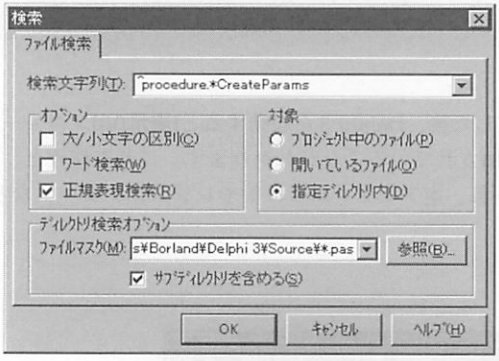
CHAP1¥CPUVIEW.DPR

Q. 複数のファイルから文字列を検索したいのですが、よい方法はないでしょうか。

Delphi 3では、**[検索(S)]｜[ファイル検索(D)]**を使うことで、プロジェクトに関連するファイルや指定したディレクトリに含まれるすべてのファイルから文字列を検索できます。

たとえば、Professional版以上に含まれているVCLソースコードからCreateParamsメソッドをオーバーライドしている部分を調べたい場合は、検索文字列として「^**procedure.***CreateParams」を入力し、**[オプション]**の**[正規表現検索(R)]**をチェックし、**[対象]**で**[指定ディレクトリ(D)]**を選び、ファイルマスクで「C:¥Program Files¥Borland¥Delphi 3¥Source¥*.PAS」を指定し、**[サブディレクトリを含める(S)]**をチェックします(図1-6)。検索文字列の「^」や「.*」は、行頭や任意の文字列のための正規表現です。正規表現の内容については、オンラインヘルプを参照してください。

図1-6　ファイル検索

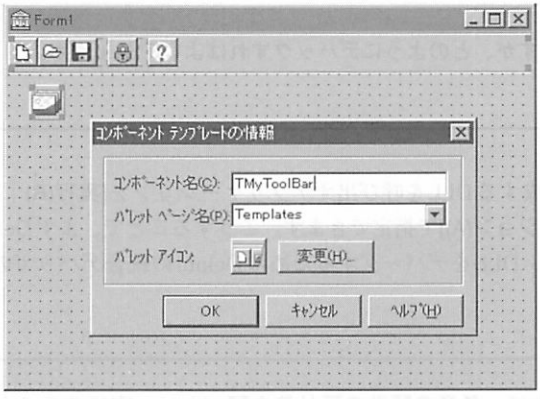


検索の結果はコードエディタのメッセージとして表示されるため、一覧表示されている項目から適当なものを選んでダブルクリックすれば、ファイルがコードエディタに読み込まれ検索された位置にカーソルが移動します。

Q. CoolBar/ToolBarやImageListを組み合わせて作成したツールバーを他のアプリケーションでも使いたいのですが、フォームをコピーする以外によい方法はないでしょうか。

Delphi 3では、コンポーネントを組み合わせたり、イベントハンドラを定義したものを再利用するためにコンポーネントテンプレートという機能があります。フォーム上に配置し、プロパティやイベントハンドラを設定したコンポーネントを選び、**[コンポーネント(C)｜コンポーネントテンプレートの作成(T)]**を呼び出すと、図1-7のようなダイアログボックスが表示され、新しいコンポーネントとして登録できます。

図1-7　コンポーネントテンプレートの登録



登録したコンポーネントは、コンポーネントパレットの指定したページ（Templatesなど）に表示され、別のフォームやアプリケーションでも利用できるようになります。イメージを登録したImageListやプロパティを設定したデータベースコンポーネントなどを登録しておけば、これらを再利用する手間が大幅に省けます。

コンポーネントテンプレートは、他のコンポーネントと異なりVCLに直接組み込まれるわけではありません。このテンプレートを他の人が利用する場合は、BINディレクトリのDELPHI32.DCTをコピーする必要があります。

Q. **複数のプログラマでアプリケーションを開発しているのですが、オブジェクトリポジトリを共有することはできませんか。**

[ツール(T)｜環境オプション(O)]の[設定]ページで、[共有リポジトリ]のディレクトリを共通のネットワークディレクトリに設定しておけば、複数のプログラマが同じリポジトリ情報を共有できます。

Delphi 2.0では、HKEY_CURRENT_USER¥Software¥Borland¥Delphi¥2.0¥Repositoryキーに BaseDirという文字列キーで共有ディレクトリを指定します。

Q. **DLLを作成しているのですが、どのようにデバッグすればよいでしょうか。**

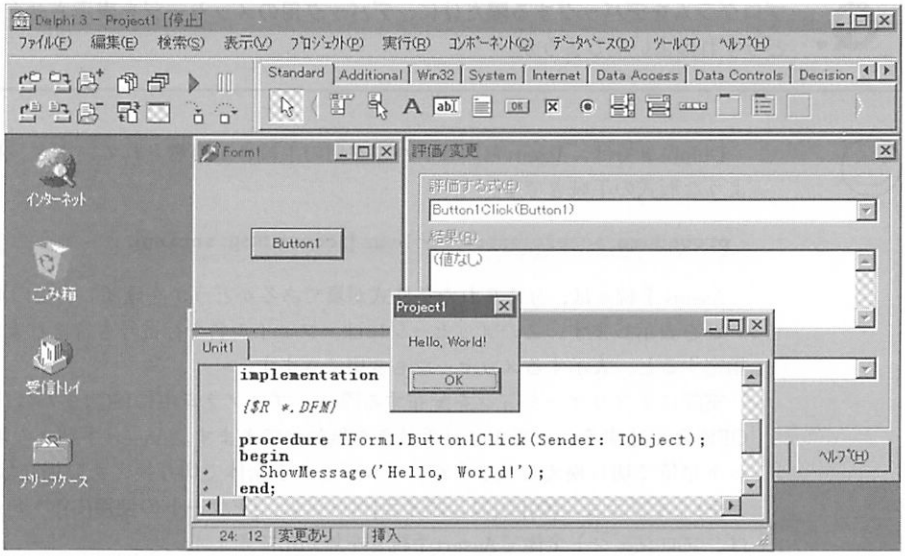
Delphi 3では、作成するDLLを呼び出すアプリケーションを[実行(R)｜実行時引数(P)]の[ホストアプリケーション(A)]で指定できます。こうすることで、まずDelphiがアプリケーションを呼び出し、DLLをデバッグするためにDelphiの統合デバッガの機能を利用できます。

Q. **プログラムのデバッグ中に、任意の関数の戻り値を調べたり、実行することはできませんか。**

[実行(R)｜評価/変更(V)]メニューや〔Ctrl〕+〔F4〕で呼び出せる式の評価を使うことで、変数を見たり、計算させる他にも、さまざまなことを試すことができます。たとえば、Form1に配置したButton1のOnClickにイベントハンドラが割り当て、Form1の別のイベントハンドラで停止中の場合、式の評価を呼び出して[評価する式(E)]として「Button1Click(Button)」と入力します。

[評価(V)]ボタンを押すと、実際にイベントハンドラが呼び出されます（図1-8）。このとき、ダイアログボックスを表示するような手続きやメソッドを使うと、開発環境とアプリケーションとの前後関係がうまく調整されないことがあります。この場合、互いのウィンドウをクリックしたり、最小化するなどしてください。また、オープン配列のように特殊な引数を持つものは評価できないこともあります。

図1-8　式の評価を使ったイベントハンドラの呼び出し



通常、プロジェクトで作成した手続きや関数にはデバッグ情報が追加されているため、このような関数は式の評価で呼び出すことができます。ただし、どの場合でもプログラムの停止した場所から呼び出せるものでなくてはなりません。たとえば、作成されていないフォームのイベントハンドラを呼び出そうとすると、エラーが発生したり、致命的な問題が生じることもあるので注意してください。また、どこからも使われていない手続きは、最適化によって削除されてしまうため、使えない場合もあります。

フルインストールするか、カスタムインストールで[デバッグライブラリファイル]を指定している場合（デフォルト）は、SLIBディレクトリにあるデバッグ情報付きのコンポーネントライブラリをLIBディレクトリにコピーすることで、VCLが提供する手続きやコンポーネントについても同じ処理ができます。この場合も、プログラムで直接または間接的に使われていない手続きなどは削除されてしまうため、呼び出せません。

式の評価で手続きを呼び出す場合、対象となるプログラムコードの実行位置が一時的に切り換えられていることに注意してください。手続きが内部の変数や状態を変更するような場合、プログラムを継続して実行する際の障害になることもあります。

Q. プログラムをデバッグする際だけに、デバッグ用のメッセージを表示させたいのですが、どうすればよいでしょうか。

Delphi 3では、Assertというデバッグ用の手続きが用意されています。Assertは、次のような形式の手続きです。

```
procedure Assert(expr: Boolean [; const msg: string]);
```

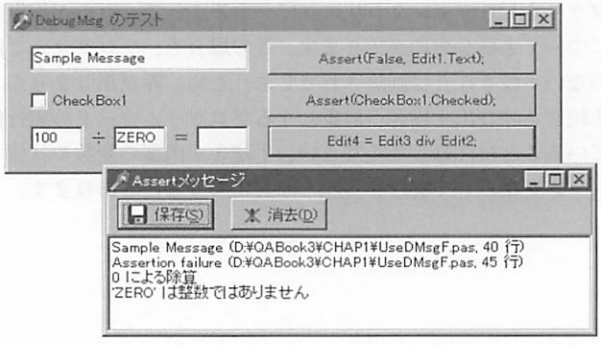
Assert手続きは、与えられた条件式が真であるかどうかを確認し、偽の場合にはメッセージを表示します。このメッセージには、ファイル名や行番号も含まれます。第2引数を指定すると、表示するメッセージも変更できます。

実際にアプリケーションを配布する際は、プログラム中に{$C-}または{$ASSERTIONS OFF}と記述することでAssert手続きを無効化できます。Assert手続きの無効化は、ユニット単位で切り換えられますが、プロジェクト全体で切り換えたい場合は、**[プロジェクト(P)｜オプション(O)]**の**[コンパイラ]**ページで、**[アサートの使用(C)]**のチェックをはずせば、プロジェクト全体でAssert手続きを無効化できます。

なお、**[ツール(T)｜環境オプション(O)]**の**[設定]**ページで、**[例外でデバッガを開く(B)]**がチェックされているときは、Assertで問題が起きるたびにプログラムが中断されます。Assertを多用する場合は、外しておく方がよいでしょう。

付録CDに収録されているDebugMsgコンポーネントは、Assert手続きで発生するEAssertionFailed例外を独自のメッセージウィンドウに表示するものです。適当なフォームにDebugMsgコンポーネントを配置すると、Assert手続きで発生する例外メッセージが独自のウィンドウに表示されるようになります。アプリケーションを配布する際は、プログラム中に{$C-}を埋め込み、DebugMsgコンポーネントを削除します。

図1-9　DebugMsgの使用例



CHAP1¥USEDMSG.DPR（DebugMsgコンポーネントを使用）

第2章

アプリケーション／フォーム

本章では、Delphiで作成するアプリケーションやフォームに関する質問について取り上げています。

Q. 一つのプロジェクトで複数のフォームを使っているのですが、あるフォームから別のフォームを表示しようとすると「未定義の識別子」というエラーになってしまいます。

Delphiでは、どんなフォームも必ずユニットというObject Pascalのソースプログラムと連携して使われます。このユニットはそれぞれ独立しているため、あるフォームから別のフォームを使う、つまりあるユニットが別のユニットを使うためにはUses節にそのユニットの名前を指定する必要があります。

たとえば、Form1とForm2を保持するUnit1とUnit2があれば、次のようにUses節の最後にUnit2を追加することでForm1からForm2を呼び出せるようになります。

```
unit Unit1;

interface;

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Unit2;
```

Delphi 3では、[ファイル(F)｜ユニットを使う(U)]メニューで目的のフォームが含まれるユニット名を指定すれば、自動的に**uses**節が追加されます。この**uses**節は**implementation**部（ソースコードの**implementation**より後ろ）に追加されます。

このようにビジュアル開発環境で、フォームとフォームの利用関係を指定した状態をフォームリンクと言い、単にフォームが呼び出せるだけでなく、リンクしたフォーム上に配置されたコンポーネントも使えるようになります。

Uses節は、C/C++で#includeを使ってヘッダファイルをインクルードすることに似ています。このユニットをコンパイルするときには、Unit2をコンパイルしたユニットオブジェクトファイル(UNIT2.DCU)というファイルを使います。もし、使うべきUNIT2.DCUがなかったり古いものであればUNIT2.PASが再コンパイルされます。なお、ユニットファイルを保存するときに名前を変更しても、Uses節に追加した名前は自動的には変更されませんので、注意してください。

Usesはユニットオブジェクトを利用しますが、別のソースプログラムをC/C++の#includeのようにそのまま取り込みたい場合は{$I filename}というコンパイラ指令を使います。たとえば、{$I MYDEFINE.INC}と記述すればMYDEFINE.INCをソースプログラムとして取り込むことができます。

コンパイラ指令についてはオンラインヘルプの「コンパイラ指令」を参照してください。

CHAP2¥FRMPROJ.DPR

Q. アプリケーションが二重に起動できないようにしたいのですが、どうすればよいでしょうか。

付録CDには、メインフォームに配置するだけで二重起動を防止できるようにするSingleInstanceコンポーネントが収録されています。SingleInstanceをメインフォームに配置すると、メインフォームが作成されるときにミューテックスを調べ、すでに作成されている場合はDupInstActionで指定された動作を実行します。

通常はdiActivateFirstになっており、最初に起動されているインスタンスを調べ、そのウィンドウをアクティブにした上でアプリケーションを終了します。diExceptionにすると、EMultiAppInstance例外を発生します。このときのメッセージには、ErrorMessageプロパティの文字列が使われます。diTerminateのときは、何もせずにアプリケーションを終了します。また、二重起動を検出するとOnDuplicateイベントが発生するため、ここで独自の処理を記述することもできます。

なお、このコンポーネントを使う場合、その性格上必ずフォームオブジェクトが生成されるため、diTerminateなどで直接終了する場合でも、一瞬フォームが表示されることがあります。これが気になる場合は、SysCtrlsユニットのCheckSingleInstance手続きを使って、プロジェクトソースを変更します。CheckSingleInstance手続きは次のような形式を持ちます。

```
procedure CheckSingleInstance(AName: string; ActivateFirst: Boolean);
```

ANameにはアプリケーション固有のユニークな文字列を与え、ActivateFirstは最初に起動されたアプリケーションをアクティブにするかどうかを指定します。例外は使われません。CheckSingleInstanceを使った例を以下に示します。

```
uses
  Forms,
  SysCtrls,
  OnlyOneF in 'OnlyOneF.pas' {Form1};
 ...
begin
  CheckSingleInstance('OnlyOneApplication', True);
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

統合開発環境からは、一つのアプリケーションしか実行できません。

CHAP2¥ONLYONE.DPR（SysCtrlsユニットを使用）

Q. アプリケーションが起動されたディレクトリパスを知るにはどうすればよいでしょうか。

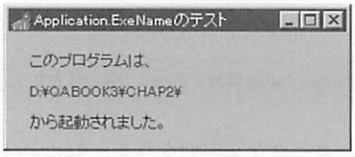
Delphiで作成するアプリケーションでは、Applicationというオブジェクトを使ってアプリケーション全体に関する情報を調べたり、イベントを処理することができます。

ApplicationのExeNameというプロパティには、アプリケーション自身のファイル名がパス付きで代入されています。ファイル名以外の起動パスを知りたい場合は、このExeNameからExtractFilePathという関数を使ってパス名を取り出します。末尾の'¥'が不要な場合は、ExtractFileDirを使います。

たとえば、アプリケーションの起動パスに同じファイル名の.INIファイルを作成したい場合は、ChangeFileExt(Application.ExeName, '.INI');とできます。たとえば、「C:¥TMP¥SAMPLE.EXE」というファイル名であれば、これによって「C:¥TMP¥SAMPLE.INI」というファイル名が得られます。

また、拡張子のないアプリケーション名を得る場合は、ChangeFileExt(ExtractFileName(Application.ExeName), '');のようにできます。

図2-1　ExeNameの使用例



ファイル名の処理については、オンラインヘルプで「ファイル管理ルーチン」を検索してください。

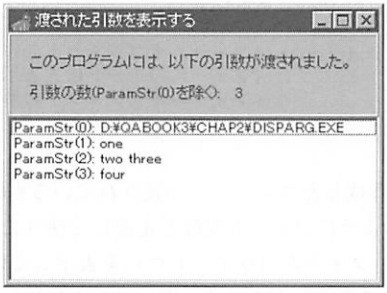
CHAP2¥EXEPATH.DPR

Q. アプリケーションに渡されるコマンドライン引数は、どうやって調べればよいでしょうか。

コマンドライン引数を調べるためには、ParamStrという関数を使います。コマンドライン引数は、スペースで区切られた文字列としてみなされ、ParamStr(1)は1番目の引数、ParamStr(2)は2番目の引数をとります。引数の数はParamCountという関数で得られます。ParamStr(0)では、アプリケーションのパス名がディレクトリつきで渡されます。これは、Application.ExeNameと同じものです。

コマンドラインはスペースで区切られます。また、ダブルクォート(")で囲まれた文字列は一つの引き数として解釈されます。たとえば、付録CDのサンプルプログラムDispArgに「one "two three" four」という引数を渡した場合は、図2-2のようになります。

図2-2　DispArgの使用例



コマンドライン引数に渡された内容をそのままの状態で調べたい場合は、CmdLineというグローバル変数を参照します。CmdLineはPChar型、つまりヌルで終わる文字列として扱わねばなりません。ヌルで終わる文字列をPascal形式の**string**型に変換するためにはStrPasという関数を使います。たとえば、Edit1.Text := StrPas(CmdLine);とすれば、コマンドライン引数をそのままEdit1コンポーネントのテキストとして代入できます。

CHAP2¥DISPARG.DPR

Q. たくさんのフォームを使っていますが、メモリを節約するために[プロジェクト(P)｜オプション(O)｜フォーム]でいくつかのフォームを[自動作成の対象(A)]から[選択可能なフォーム(F)]に変更しました。この[選択可能なフォーム(F)]は、どのように使えばよいでしょうか。

メインメニューの[表示｜プロジェクトソース(J)]で表示されるプロジェクトソースを見ると、[自動作成の対象]となっているフォームに対してはApplication.CreateForm(TForm1, Form1);という文があり、この文によってアプリケーションが実行する(Application.Run;) 前にフォームが作成されます。

選択可能なフォームにした場合はこの場所ではフォームは作成されなくなります。このため、フォームが必要になった時点で明示的にフォームを作成します。たとえば、Form1のButton1を押したときにUnit2で定義されているForm2を表示するためには、次のように記述します。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2 := TForm2.Create(Application); {フォームオブジェクトの作成}
  Form2.ShowModal;                    {フォームのモード付き表示}
  Form2.Release;                      {フォームオブジェクトの解放}
end;
```

Form2という変数は、フォームを作成したユニットで定義されているものです。このメソッドの中で**var** Form2: TForm2;のようにフォーム変数を定義して使うこともできます。

このモード付き(Modal)表示とは、フォーム（ウィンドウ）を表示している間はアプリケーションの他のウィンドウに移動できないものです。ShowModalはフォームを閉じるまではShowModalメソッドから戻りません。逆に、ShowModalから戻ってきた時点ですでにフォームの処理は終わっているためフォームが確保していたメモリをReleaseで解放できます。

Showメソッドを使ったモードなし表示の場合、フォームを表示するとすぐに処理が戻ってくるため、ここでフォームのメモリを解放してはいけません（フォームのメモリを解放すると、直ちにフォームが閉じられます)。この場合は、別のアクション（閉じるためのボタンを押すなど）でフォームを解放するように記述します。このプログラムでは、クローズボックスをダブルクリックしてForm2を閉じた場合には、ウィンドウは非表示状態になりますがメモリは解放されません。メモリが解放されるのは、Button2を押したときです。

```
{ Button1 を押すと、Form2 を表示する }
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Form2 = nil then
    Form2 := TForm2.Create(Application);
  Form2.Show;
end;

{ Button2 を押すと、Form2 を閉じる }
procedure TForm1.Button2Click(Sender: TObject);
begin
  if Form2 <> nil then
  begin
    Form2.Release;
    Form2 := nil;
  end;
end;
```

このプログラムでは、イベントハンドラの中にForm2変数を定義してはいけません。Form2変数を定義すると、別ユニットで定義されているForm2変数を隠してしまうため、異なるイベントハンドラで同じ変数を参照できなくなってしまうためです。

この方法では、同じフォームクラスからいくつかのフォームを表示することができません。複数のフォームを作成して、フォームを閉じた時点でメモリを解放するには、フォームのOnCloseイベントハンドラを定義します。

```
{ Form2 を閉じるときに自動的にメモリを解放する }
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

こうしておけば、フォーム変数を使ってフォームを管理する必要もなくなります。フォームを作成するのは次のようにするだけです。

```
{ Button1 を押すたびに、新しいフォームを作成する }
procedure TForm1.Button1Click(Sender: TObject);
begin
  with TForm2.Create(Application) do { フォームオブジェクトの作成 }
    Show;                            { フォームのモード付き表示 }
end;
```

CHAP2¥DYNFORM.DPR

Q. アプリケーションを起動するときに、メインフォームのOnCreateイベントハンドラで時間のかかる初期設定をしています。この間に、オープニングダイアログボックスを表示したいのですが、別のフォームを表示しようとするとエラーになってしまいます。どうすればよいでしょうか。

まず、メインフォームのOnCreateイベントハンドラがいつ呼び出されるのかを知る必要があります。このために、まず2つのフォームを持つプロジェクトを作ります。アプリケーションを新規に作成し、さらに空のフォームを追加します。

ここで、メインメニューから[表示(V)｜プロジェクトソース(J)]を選んでください。プロジェクトソースは次のようなものになっているはずです。

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.PAS' {Form1},
  Unit2 in 'Unit2.PAS' {Form2};

{$R *.RES}

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

ここで、**begin**～**end**の間がこのプロジェクトのメインプログラムです。通常、DelphiではApplicationというオブジェクトがアプリケーション全体を管理します。

まず、ApplicationのCreateFormというメソッドを使ってフォームの実体（オブジェクト）を生成します。CreateFormが使われるのは、[プロジェクト(P)｜オプション(O)｜フォーム]ページの[自動生成の対象(A)]にリストアップされているものだけです。[選択可能なフォーム(F)]というのは、使うときに自分でフォームの実体を生成する必要があるものです。また、最初にCreateFormで生成されたフォームがアプリケーションのメインフォームになります。そして、Runメソッドでアプリケーションを実行させます。

最初の問題で、Form1のOnCreateイベントが発生するのは、このApplication.CreateForm(TForm1, Form1);が呼び出されたときです。たとえ、プロジェクトオプションで自動生成するようにしていても、この時点ではForm2は生成されていないのです。Form1のOnCreateイベントハンドラで、Form2.Show;などとしても生成されていないフォームを使うことになるのでエラーが発生します。

Form1とForm2の作成順序を入れ換えればよいように見えるかもしれませんが、CreateFormでは最初に呼び出されるものがメインフォームになるので、期待通りにはな

りません。メインフォームのOnCreateイベントハンドラで呼び出すフォームは、プロジェクトソースでCreateFormを使うべきではないのです。

元の課題を実現するためには、まず[プロジェクト(P)｜オプション(O)｜フォーム]ページでForm2（オープニングダイアログに使いたいフォーム）を[選択可能なフォーム(F)]移しておきます。こうすることで、プロジェクトソースからForm2のためのCreateFormの呼び出しがなくなります。次に、Form1のOnCreateイベントハンドラに次のプログラムを割り当てます。このとき、[ファイル(F)｜ユニットを使う(U)]でForm2が定義されているUnit2を選んでおきます。

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form2 := TForm2.Create(Application); {プログラムでフォームを生成}
  Form2.Show;                          {フォーム自身の表示}
  Form2.Update;                        {フォーム全体の更新}
  { 時間のかかる初期設定などの処理 }
  Form2.Release;                       {フォームの解放}
end;
```

もし、初期設定の進行状況をフォームに反映させたり、初期設定中も他のアプリケーションに切り換えられるようウィンドウメッセージを処理したいのであれば、処理の途中でApplication.ProcessMessages;を呼び出すようにします。進行状況をフォームに表示させて、他のアプリケーションに切り換えられたり、ウィンドウメッセージを受け付けたくない場合には、値が変更された時点でForm2.Update;を呼び出します。

Form2にコンポーネントパレットのSamplesページにあるGaugeコンポーネントが配置してあるとき、次のようにプログラムすると進行状況をゲージに表示できます。進行状況は1%単位で表示する必要はありません。

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  Form2 := TForm2.Create(Application); {プログラムでフォームを生成}
  Form2.Show;                          {オープニングフォームの表示}
  Form2.Update;                        {フォーム全体の更新}
  for i := 0 to 100 do                 {ゲージの範囲に合わせる}
  begin
    {時間のかかる初期設定処理}
    Form2.Gauge1.Progress := i;        {ゲージの進行状況を設定}
    Form2.Update;                      {フォームの更新}
  end;
  Form2.Release;                       {フォームの解放}
end;
```

また、プロジェクトソース(.DPR)そのものを変更することもできます。やはりオープニングダイアログとして使いたいフォームを[自動生成の対象(A)]から[選択可能なフォーム(F)]に変更しておきます。オープニングフォームの表示をメインフォームのOnCreateに

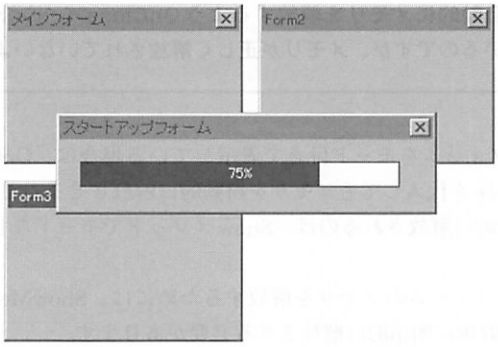
記述する代わりにプロジェクトソースに直接記述します。

前述と同様にオープニングフォームにGaugeコンポーネントを配置しておき、残りのフォームの数をMaxValueプロパティに合わせておき、CreateFormでフォームを作成するごとにGaugeコンポーネントのProgressに値を設定しておけば、フォームが作成されるたびにゲージを更新できます。

5つのフォームを作成してForm5をオープニングフォームとして使う場合、次のようにプログラミングできます。ここで、Form5にはGaugeコンポーネントを配置しMaxValueに4を設定しておきます。また、実行している状態を図2-3に示します。

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.PAS' {Form1},
  Unit2 in 'Unit2.PAS' {Form2},
  Unit3 in 'Unit3.PAS' {Form3},
  Unit4 in 'Unit4.PAS' {Form4},
  Unit5 in 'Unit5.PAS' {Form5};

{$R *.RES}

begin
  Application.Initialize;
  { オープニングダイアログとして使いたいフォーム(Form5)を構築する }
  Form5 := TForm5.Create(Application);
  Form5.Show;
  Form5.Gauge1.Progress := 0;
  Form5.Update;
  { フォームの作成と、オープニングダイアログのゲージの更新を繰り返す }
  Application.CreateForm(TForm1, Form1);
  Form5.Gauge1.Progress := 1;
  Form5.Update;
  Application.CreateForm(TForm2, Form2);
  Form5.Gauge1.Progress := 2;
  Form5.Update;
  Application.CreateForm(TForm3, Form3);
  Form5.Gauge1.Progress := 3;
  Form5.Update;
  Application.CreateForm(TForm4, Form4);
  { オープニングダイアログの最後の更新と解放 }
  Form5.Gauge1.Progress := 4;
  Form5.Update;
  Form5.Release;
  Application.Run;
end.
```

図2-3　スプラッシュ画面の表示



ただし、プロジェクトソースをこのように変更すると、新しいフォームを作成して追加するとForm1の後ろに新たなフォームを作成する行が挿入されます。Delphiは、プロジェクトソースを自動的に管理していますが、Application.CreateForm(フォームクラス名,フォーム名);というスタイルが連続しているものをプロジェクトで自動生成するフォームとして認識しているためです。プロジェクトソースを変更する場合には、このような点に注意が必要です。

CHAP2¥PROGRESS.DPR
CHAP2¥SPLASH.DPR

Q. フォームを閉じるときに、自動的にメモリを解放するようOnCloseイベントハンドラでActionにcaFreeを代入しているのですが、メモリが正しく解放されていないようです。

ShowModalを使ってフォームをモード付きで表示している場合は、OnCloseイベントハンドラでActionにcaFreeを代入してもメモリを自動的に解放させることはできません。フォームのメモリが自動的に解放されるのは、Showメソッドでモードなし表示した場合だけです。

モード付き表示させたフォームのメモリを解放するためには、ShowModalメソッドの呼び出しから戻ってきた直後に明示的に解放させる必要があります。

なお、フォームがfsMDIChildスタイルの場合は、OnCloseでActionにcaFreeを代入しておけばフォームを閉じるときにアイコン化されず、完全に閉じてメモリが解放されます。fsMDIChildではActionにcaHideを指定することはできません。

通常、MDIの子フォームは非表示状態にできませんが、どうしても非表示にさせたい場合は直接APIを使ってShowWindow(ActiveMDIChild.Handle, SW_HIDE);のようにします。この場合、ユーザー操作でウィンドウを破棄できなくなりますので、子ウィンドウを再表示したり破棄するためには、プログラムで非表示にした子ウィンドウをきちんと管理しておく必要があります。

CHAP2¥MDIPROG.DPR

Q. アプリケーションの空いている時間を使うためのOnIdleイベントや、ヒントメッセージをステータスバーに表示するためOnHintイベントを使いたいのですが、これらはフォームのイベントのようにビジュアルに設定できないのでしょうか。

アプリケーション（Application）は背後で動作するため、オブジェクトインスペクタを使ってプロパティやイベントを設定することはできません。

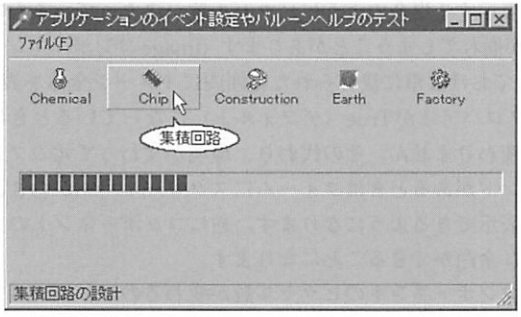
付録CDにはアプリケーションの状態を設定するためのPseudoAppコンポーネントが収録されています。PseudoAppには、ほぼApplicationと同じプロパティやイベントが定義されており、実行時には設定した内容がApplicationオブジェクトに反映されます。

CHAP2¥USEPAPP.DPR（PseudoApp、CoolHintコンポーネントを使用）

Q. Hintプロパティを使ったヒント表示を長方形以外の形にすることはできませんか。

付録CDには、ヒント表示の形状を変更するためのCoolHintコンポーネントが収録されています。CoolHintは、バルーンヘルプ（図2-4）や角の丸い長方形スタイルのヒント表示を設定できます。また、自分でイベントハンドラを定義すれば、自由なスタイルのヒント表示を実現できます。

図2-4　バルーンヘルプ



CHAP2¥USEPAPP.DPR（PseudoApp、CoolHintコンポーネントを使用）

Q. あるマシンで作成したフォームを別のマシンで実行したところ、フォームが期待する位置に表示されませんでした。また、フォームに配置したコンポーネントがフォームに比べて大きくなってしまい、フォームにスクロールバーが付くことがあります。コンポーネントの大きさに比例させてフォームの大きさを変えるにはどうすればよいでしょうか。

通常、フォームの表示位置や大きさは、設計時と同じになっています。たとえば、1024×768の解像度で設計したフォームを640×480の解像度のマシンで実行すると、元のフォームの位置によっては、フォームが見えない場所に配置される可能性があります。

もし、フォームを常に画面中央に表示させたいということであれば、フォームのPositionプロパティをpoScreenCenterに設定します。また、サイズを変更したくないが必ず見える場所に表示したいという場合は、poDefaultPosOnlyを指定します。位置も大きさも任意でかまわない場合は、poDefaultを指定します。

設計時と同じサイズを使っている場合でも、設計時と実行時の解像度やフォントサイズの違いによって、フォーム上のコンポーネントの大きさが変わって、フォームにスクロールバーが付けられてしまうことがあります。スクロールバーをつけないようにするには、AutoScrollプロパティをFalseにするだけです。また、少し制約はありますがScaledプロパティをFalseにするという方法もあります。Delphiのフォームと配置したコンポーネントの大きさの関係をよく知るために、これらのプロパティについて説明します。

リソースエディタと呼ばれるツールで作成するWindowsのダイアログボックスでは、デフォルトでシステムフォントが使われるため解像度やフォントのドット数が変更されると、ダイアログボックスや配置しているコントロールの高さや幅も変わります。

これと同じように、Delphiのフォームでも使われているフォントの大きさが変わると、フォームに配置しているコンポーネントの大きさが変わります。このため、Imageコンポーネントでビットマップを表示する場合のようにピクセル数が決まっているものでは、表示される大きさのバランスが崩れてしまうことがあります（Imageコンポーネントでは、Stretchプロパティを Trueにしておけば常に決められた範囲内にイメージ全体を表示します）。

しかし、AutoScrollプロパティがTrue（デフォルト）になっているときは、フォームそのもののピクセル数は変わりません。その代わり、環境が変わって元のフォームからはみ出すようなコンポーネントがあるときはフォームにスクロールバーがつき、スクロールバーでコンポーネントを表示できるようになります。逆にコンポーネントの大きさが小さくなるときはフォーム上に余白ができることになります。

このように自動的にコンポーネントのピクセル数が変わるのは、Scaledプロパティがデフォルトで Trueになっているためです。もし、Scaledプロパティを Falseにすると、そのフォームは解像度が違う別の環境で実行しても常に設計されたときと同じピクセル数で表示されます。このため、ほとんどの場合はスクロールバーが付きません。

ただし、フォームの大きさが変わらないのに対し、メニューの高さはシステムフォントの大きさに依存するため、若干クライアント領域の大きさが小さくなることはあります。

フォームの下端にコンポーネントを配置していたり、メニューの高さが大きく変わる場合には、スクロールバーが付いてしまうことがあります。また、解像度が低いマシンから高いマシンへ移行すると極端にフォームが小さくなってしまうことがあります。これが、Scaledプロパティを Falseにする場合の制約です。

Scaledプロパティが Trueでも、AutoScrollプロパティを Falseにしておけば、フォームにスクロールバーがつかない代わりに、コンポーネントの大きさ（フォントのピクセル数）に比例して、フォーム全体のピクセル数が変更されます。AutoScrollプロパティが False、Scaledプロパティが Trueという状態にしておけば、通常のダイアログボックスと同じようになると言えます。

また、たとえばBorderStyleを bsSizeableか bsSizeToolWin以外に設定すると、自動的にAutoScrollが Falseになります。bsSizeableや bsSizeToolWinのときは自分でウィンドウの大きさを変更してスクロールバーが不要になるように調整できますが、それ以外のときはウィンドウの大きさを変更できないためです。BorderStyleを設定した後でAutoScrollを設定することもできますが、あまり好ましくないでしょう。

もし、画面の解像度に関係なくスクリーンと同じ大きさや比率を保ちたいという場合はScaleByというメソッドを使えます。ScaleBy(M, D);という呼び出しは、現在のフォームの大きさをM/D倍します。たとえば、フォームの横幅をスクリーンの大きさに合わせたい場合は、ScaleBy(Screen.Width, Width);とします。

CHAP2¥FMSCALE.DPR

Q. **フォームのAutoScrollプロパティを Trueにして、表示できないコンポーネントをスクロールバーで表示できるようにしているのですが、フォーム上に何かを描画しようとするとスクロールの状態に関係なく同じ位置に描画されてしまいます。フォームの内容がどれだけスクロールしているかを調べるには、どうすればよいでしょうか。**

スクロールバーの情報を考慮する場合には、水平座標は HorzScrollBar.ScrollPos + X、垂直座標は VertScrollBar.ScrollPos + Yとします。

このHorzScrollBar（水平）と VertScrollBar（垂直）というプロパティは、フォームに貼り付けられるスクロールバーを管理するものです。フォームのAutoScrollプロパティが Trueで、配置したコントロールがはみ出している場合は、自動的に必要なスクロールバーの Visibleプロパティが Trueになり、Range（範囲）が設定されます。スクロールバーが表わす位置は、Positionというプロパティで設定したり参照できますが、実行時にはScrollPosというプロパティを参照する方がよいでしょう。ScrollPosは、スクロールバーが表示されていないときには0を返します。

CHAP2¥SCRLPOS.DPR

Q. アプリケーションを終了するときにフォームの位置や大きさをレジストリに記録しておき、次に起動したときに同じ場所に表示させたいのですが、どのようにプログラムすればよいでしょうか。

付録CDには、フォームの位置をレジストリに保存したり、復元できるようにするWinPlaceコンポーネントが収録されています。通常は、フォームの位置を記録しておくためのレジストリキーとしてHKEY_CURRENT_USER¥Software¥DelphiApp¥<プロジェクト名>が使われます。KeyNameプロパティを使えば、¥Software以降に独自のキー名を指定できます。複数のフォームの位置を記録する場合はフォームごとにWinPlaceを配置しますが、キー名は必ず共通になります。データが記録されるエントリ名はフォーム名＋コンポーネント名（例: Form1WinPlace1）になり、これは変更できません。

なお、実際にフォーム位置を保存・復元するためにはReadInfo、WriteInfoメソッドを使います。典型的な使い方では、フォームのOnCreateとOnDestroyイベントハンドラを次のように定義します。

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  WinPlace1.ReadInfo;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  WinPlace1.WriteInfo;
end;
```

CHAP2¥USEWPLC.DPR（WinPlace、FontInfoコンポーネントを使用）

Q. アプリケーションを終了するときにフォームやコントロールのフォントをレジストリに記録しておき、次に起動したときに同じフォントを使いたいのですが、どのようにプログラムすればよいでしょうか。

付録CDには、フォントをレジストリに保存・復元できるようにするFontInfoコンポーネントが収録されています。通常は、フォントを記録しておくためのレジストリキーとしてHKEY_CURRENT_USER¥Software¥DelphiAppが使われます。KeyNameプロパティを使えば、¥Software以降に独自のキー名を指定できます。複数のフォントを記録する場合は複数のFontInfoを配置しますが、キー名は必ず共通になります。データが記録されるエントリ名はEntryNameプロパティ名で指定します。デフォルトはコンポーネント名と同じです。

なお、実際にフォントを保存・復元するためにはReadFont、WriteFontメソッドを使います。ReadFontではレジストリが見つからなかったときのためのデフォルトのフォントを指定します。典型的な使い方では、フォームのOnCreateとOnDestroyイベントハンドラを次のように定義します。

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Font := FontInfo1.ReadFont(Font);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  FontInfo1.WriteFont(Font);
end;
```

CHAP2¥USEWPLC.DPR（WinPlace、FontInfoコンポーネントを使用）

Q. **カーソル形状を変更したいのですが、フォームのCursorプロパティにcrHourGlassなどを代入しても、何も変わりません。**

カーソル形状はコンポーネントごとに指定されています。このため、パネル(Panel)など他のコンポーネントがフォームを覆っていると、フォーム自身のCursorプロパティを変更しても何も変わらないことになります。したがって、カーソルを変更したい場所のコンポーネントのCursorプロパティを変更する必要があります。

一度にすべてを変更する簡単な方法として、フォームではなくScreenオブジェクトのCursorプロパティを変更できます。Screen.Cursor := crSQLWait;とすると、アプリケーションに関係するすべてのウィンドウのカーソルを変更できます。元に戻すときは、Screen.Cursor := crDefault;とするだけです。

カーソルを一時的に消去したい場合は、CursorプロパティにcrNoneを代入します。カーソルの形状を変更するのと同様に、消去されるカーソルもコンポーネント単位になります。アプリケーションのすべてのウィンドウでカーソルを消去するときは、Screen.Cursor := crNone;とし、元に戻すときはScreen.Cursor := crDefault;とします。

Delphiで用意されているカーソルのイメージについては、オンラインヘルプ「Cursorプロパティ（すべてのコントロール用)」を参照してください。

CHAP2¥CHGCSR.DPR

Q. 既存のマウスカーソルでなく、独自に作成したものを使いたいのですが、どうすればよいでしょうか。

Delphiのフォームでは、Windowsにあらかじめ用意されているカーソルに加えて、crSQLWait、crMultiDrag、crVSplit、crHSplit、crNoDrop、crDragという6種類のカーソル形状を新たに使えます。たとえば、あるフォーム上でこのカーソルを使いたい場合は、フォームのイベントハンドラなどでCursor := crSQLWait;のように記述するだけです。

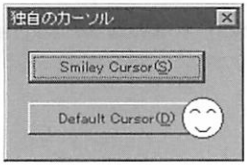
これら以外の独自のカーソルを使いたい場合は、Delphiのメニューから[ツール(T)｜イメージエディタ]を呼び出して新たなリソース(.RES)を作成し、新規リソースとしてカーソルを作成します。このとき、リソースファイル名にプロジェクト名と同じものををつけないようにしてください。プロジェクトファイル名と同じ名前のリソース(*.RES)は、Delphiがアプリケーションのために自動生成するものです。このため、この名前でリソースを作成すると、作成した内容はDelphiに上書きされてしまいます。

リソースを作成したら、プロジェクトかフォームユニットの**implementation**部で{$R filename}というコンパイラ指令を使って、リソースを組み込むようにします。たとえば、フォームユニットの**implementation**部で{$R *.RES}としておけば、フォーム名に対応する.RESファイルを読み込みます。リソースは最終的には一つに結合されるため、他のフォームやプロジェクトソースで組み込んだリソースでも同じアプリケーションの中で自由に利用できます。

独自のカーソルを使うためには、あらかじめScreenオブジェクトのCursorsプロパティを使ってカーソルを登録しておく必要があります。たとえば、プロジェクトソースやフォームの**initialization**部などでScreen.Cursors[1] := LoadCursor(HInstance, 'CURSOR_1');とします。Cursorsのインデックスには、アプリケーション全体で共通に使う1以上の値を与えます。よりわかりやすくするために**interface**部に**const**定義を追加して、crで始まるシンボルを割り当てておくとよいでしょう。また、'CURSOR_1'は新たに定義したカーソルリソースの名前です。

こうしておけば、フォームのイベントハンドラなどでCursor := 1;とするだけで、1番目のカーソル（つまり作成したカーソル）を使えるようになります。

図2-5　独自のカーソルを使う例



CHAP2¥ORIGCSR.DPR

Q. フォームのIconプロパティを指定していますが、タイトルバーの左端に表示されるアイコンが変わるだけで、タスクバーやプログラムグループに表示されるアイコンが変わりません。どうすれば、プログラムグループに表示されるアイコンを変更できるでしょうか。

アプリケーション自身のアイコンは、統合開発環境の[プロジェクト(P)｜オプション(O)]の[アプリケーション]ページでアイコンを指定します。もし、アイコンを変更してしまった後に、元に戻したい場合はDelphiをインストールしたディレクトリのImages¥Defaultディレクトリにある Default.Ico を指定し直してください。

CHAP2¥APPICON.DPR

Q. タイトルバーのないフォームは、どのように作成すればよいのでしょうか。

フォームのBorderStyleプロパティをbsNoneにします。

この設定によってタイトルバーだけでなく枠もなくなってしまいますが、タイトルバーのないフォームで枠を表示したい場合には、フォームのCreateParamsというメソッドをオーバーライドします。具体的には次のように記述できます。

```
type
  TForm1 = class(TForm)
  private
    { Private 宣言 }
  protected
    procedure CreateParams(var Params: TCreateParams); override;
  public
    { Public 宣言 }
  end;

  ...

procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  Params.Style := Params.Style or WS_BORDER; { または or WS_THICKFRAME }
end;
```

CHAP2¥NOTITLE.DPR

Q. タイトルバーのないフォームなどで、クライアント領域をクリック＆ドラッグしてフォームを移動させたいのですが、どうすればよいでしょうか。

これは、フォームを移動させるというシステムコマンドをフォーム自身に与えることで簡単に実現できます。具体的には、次のようなプログラムになります。

```
type
  TForm1 = class(TForm)
  private
    { Private 宣言 }
  protected
    procedure WMLButtonDown(var Message: TWMLButtonDown);
        message WM_LBUTTONDOWN;
  public
    { Public 宣言 }
  end;

  ...

procedure TForm1.WMLButtonDown(var Message: TWMLButtonDown);
begin
  SendMessage(Handle, WM_SYSCOMMAND, SC_MOVE or 2, 0);
end;
```

WM_SYSCOMMANDは、システムコマンドをあらわすWindowsメッセージです。メッセージの種類は、SC_xxxというシンボルであらわされ、SC_MOVE以外にSC_SIZE、SC_MINIMIZE、SC_NEXTWINDOWなどがあります。SC_xxxは、下位4ビット（16進の1桁）を内部で使用しています。SC_MOVE **or** 2としているのは2がマウスで移動する際の情報をあらわしているためです（SC_MOVEだけの場合は、キーボードでの移動になります）。第3引数をSC_SIZE **or** 8とすれば、枠がbsSizeableでない場合でもマウスでドラッグしてフォームの大きさを変更できます。

これらの具体的な数値は、Windows APIのオンラインヘルプなどには記載されていませんが、自分でウィンドウを作成したりWM_SYSCOMMANDメッセージを捕らえることで調べられます。

CHAP2¥NOTITLE.DPR

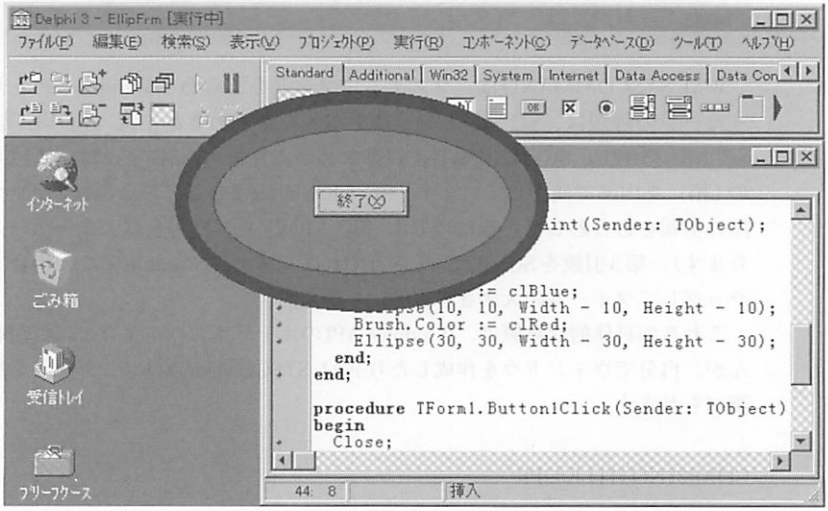
Q. 長方形でないフォームを作成することはできませんか。

Windows 95では、ウィンドウに「リージョン」（領域）を割り当てられます。リージョンには長方形の他に楕円形や多角形、あるいはこれらの組み合わせを指定できます。

たとえば、フォームのOnCreateイベントを次のように定義すれば、楕円形のフォームが作成できます。このとき、フォームのBorderStyleはbsNoneにしておきます。BorderStyleが他の値になっている場合は、タイトルバーを含めて楕円形になります。

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Rgn: HRGN;
begin
  Rgn := CreateEllipticRgn(0, 0, Width, Height);
  SetWindowRgn(Handle, Rgn, True);
end;
```

図2-6　長方形でないウィンドウ



CHAP2¥ELLIPFRM.DPR

Q. アプリケーションを実行中に、Windowsが終了しようとしているかどうかを知るにはどうすればよいでしょうか。

Windowsを終了するときも、フォームを閉じるときと同じようにOnCloseQueryイベントが発生するので、対応するイベントハンドラを記述しておけばよいでしょう。

もし、フォームを閉じる場合でなくWindowsが終了しようとする場合だけを捕らえたいのであれば、ウィンドウに渡されるWM_QUERYENDSESSIONというメッセージを処理します。ウィンドウに渡されるメッセージは、**message**という予約語を使って次のように処理できます。

```
type
  TForm1 = class(TForm)
    ...
  private
    procedure WMQueryEndSession(var Msg: TWMQueryEndSession);
              message WM_QUERYENDSESSION;
    ...
  end;

implementation

procedure TForm1.WMQueryEndSession(var Msg: TWMQueryEndSession);
begin
  Msg.Result := Longint(MessageDlg('Windows: 終了してもよろしいですか?',
                mtConfirmation, [mbYes, mbNo], 0) = mrYes);
end;
```

WM_QUERYENDSESSIONは、Windowsが終了できるかどうかを問い合わせるものです。終了することが決まった場合の処理にはWM_ENDSESSIONメッセージを使います。

CHAP2¥ENDSESS.DPR

Q. 動かせないフォームを作成したいのですが、どうすればよいでしょうか。

BorderStyleをbsNoneにしてフォームを移動するためのタイトルバーを表示させないようにする以外、フォームの位置を固定するためのプロパティはありません。しかし、Windowsのメッセージを処理することで移動できないウィンドウを作成することができます。ウィンドウを移動しようとするとWM_SYSCOMMANDというメッセージが送られますが、これを無視するようにプログラムします。

```
type
  TForm1 = class(TForm)
    CheckBox1: TCheckBox;
    procedure FormCreate(Sender: TObject);
  private
    { フォームに送られる WM_SYSCOMMAND メッセージを処理する }
    procedure WMSysCommand(var Msg: TWMSysCommand);
      message WM_SYSCOMMAND;
  end;

  ...

procedure TForm1.WMSysCommand(var Msg: TWMSysCommand);
begin
  { チェックボックスがチェックされていないときは移動できない }
  if ((Msg.CmdType and $FFF0)=SC_MOVE) and not CheckBox1.Checked then
    Msg.Result := 0       { SC_MOVE (移動)コマンドを無効化する }
  else
    inherited;            { それ以外は、デフォルトの動作 }
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  { フォームが作成されるときに右上に配置する }
  Top := 0;
  Left := Screen.Width - Width;
end;
```

CHAP2¥FIXPOS.DPR

Q. Delphiのメインウィンドウ（スピードバー／コンポーネントパレット）のように幅だけを変更でき、高さを変更できないフォームを作成するにはどうすればよいでしょうか。

WindowsメッセージのWM_GETMINMAXINFOを処理します。WM_GETMINMAXINFOは、フォームの最小、最大トラッキングサイズを格納する構造体MINMAXINFOへのポインタが渡されます。高さを変更したくない場合は、最小・最大の高さを同じ値にします。

マウスカーソルがウィンドウの上下の端にかかったときに、リサイズ用のカーソルにならないようにするためには、WM_NCHITTESTメッセージを処理します。

以下にプログラム例を示します。

```
type
  TForm1 = class(TForm)
  private
    procedure WMGetMinMaxInfo(var Msg: TWMGetMinMaxInfo);
              message WM_GETMINMAXINFO;
    procedure WMNCHitTest(var Msg: TWMNCHitTest);
              message WM_NCHITTEST;
  end;

    ...

procedure TForm1.WMGetMinMaxInfo(var Msg: TWMGetMinMaxInfo);
begin
  inherited;
  with Msg.MinMaxInfo^ do
  begin
    ptMinTrackSize.y := Height;
    ptMaxTrackSize.y := Height;
  end;
end;

procedure TForm1.WMNCHitTest(var Msg: TWMNCHitTest);
begin
  inherited;
  with Msg do
    case Result of
    HTTOP, HTBOTTOM: Result := HTNOWHERE;
    HTTOPLEFT, HTBOTTOMLEFT: Result := HTLEFT;
    HTTOPRIGHT, HTBOTTOMRIGHT: Result := HTRIGHT;
    end;
end;
```

CHAP2¥RESIZEW.DPR

Q. MDIアプリケーションを作成していますが、フォームのOnPaintで描画しても反映されません。また、LabelやImageを配置しても設計時には表示されるのに、実際に実行すると表示されなくなります。

フォームのFormStyleプロパティをfsMDIFormにして、MDIのメインフォームとして実行される場合、このフォームのクライアント領域（メニューや枠などを除いた領域）には、MDIの処理を担当する専用のクライアントウィンドウが配置されます。このため、メインのフォームにはクライアント領域の描画は要求されず、OnPaintでは何も描画できません。

付録CDには、MDIのクライアント領域にも描画できるようにするためのMDIBackコンポーネントが収録されています。

MDIBackコンポーネントをMDIのメインフォームに配置すると、OnPaintイベントが発生するようになり、Canvasプロパティを使ってクライアント領域に描画できるようになります。なお、子ウィンドウの配置によって描画内容が勝手にスクロールしないよう、スクロールバーの表示は抑止されます。また、PanelやStatusBarなどAlignプロパティをalNone以外に設定したコンポーネントを配置している場合、設計時の表示と実行時の表示がずれる場合があります。

図2-7　MDIの背景に描画する例

CHAP2¥MDIDRAW.DPR（MDIBackコンポーネントを使用）

Q. **MDIアプリケーションを作成していますが、スクロールバーを表示させないようにするにはどうすればよいでしょうか。**

前項目で説明したMDIBackコンポーネントを配置することで、スクロールバーの表示は抑止されます。

CHAP2¥MDIDRAW.DPR（MDIBackコンポーネントを使用）

Q. **MDIフォームのWindowMenuを指定して、あるメニュー項目にウィンドウの一覧を表示させているのですが、このメニュー項目に新しいサブメニューを追加するとウィンドウの一覧がなくなってしまいます。**

MDIアプリケーションにおいて、メニューによるウィンドウリストなど多くの処理はWindows自身が提供するクライアントウィンドウにまかされています。外部でウィンドウメニューを変更した場合は、このことをクライアントウィンドウに通知しなければなりません。このために、WM_MDIREFRESHMENUというメッセージを使います。たとえば、次のようにプログラムします。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Item: TMenuItem;
begin
  Item := TMenuItem.Create(Window1);
  Item.Caption := Edit1.Text;        { 新しいメニュー項目名 }
  Item.OnClick := NewCommand;        { イベントハンドラの設定 }
  Window1.Insert(2, Item);
  SendMessage(ClientHandle, WM_MDIREFRESHMENU, 0, 0);
end;
```

CHAP2¥MDIDRAW.DPR（MDIBackコンポーネントを使用）

Q. [プロジェクト(P)｜オプション(O)]の[アプリケーション]ページでヘルプファイルを指定しているのですが、通常のフォームでは〔F1〕キーを押すとヘルプファイルが表示されるのに、MDIフォームの場合は何も表示されません。

MDIフォームでは、〔Ctrl〕+〔F6〕による子フォームの切り替えなどの処理をするためにキー入力をMDICLIENTという特殊なウィンドウが処理しています。アクティブな子フォームやコントロールがない場合は、MDIアプリケーションへのキー入力はMDICLIENTウィンドウに送られます。このウィンドウはWindows自身が提供するものなので、DelphiのフォームのOnKeyDownイベントなどでは処理できません。

MDICLIENTに送られるキー入力を処理するためには、Applicationオブジェクトの OnMessageイベントを使います。このイベントは、Windowsメッセージが送られるたびに発生するもので、アプリケーションに送られるすべてのメッセージを処理できます。

MDIアプリケーションで〔F1〕キーを押したときにヘルプファイルを表示するには、次のように記述します。

```
type
  TForm1 = class(TForm)
    ...
    procedure FormCreate(Sender: TObject);
  private
    procedure AppMessage(var Msg: TMsg; var Handled: Boolean);
  end;
  ...
{ フォーム作成時に、Application の OnMessage イベントハンドラを設定 }
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage = AppMessage;
end;

{ OnMessage イベントハンドラ }
procedure TForm1.AppMessage(var Msg: TMsg; var Handled: Boolean);
begin
  with Msg do
    if (hwnd = ClientHandle) and (message = WM_KEYDOWN)
       and (wParam = VK_F1) then
    begin
      Application.HelpContext(HelpContext);
      Handled := True;
    end;
end;
```

CHAP2¥MDIHELP.DPR

Q. 作成するアプリケーションにエクスプローラからファイルをドラッグ＆ドロップしたいのですが、どうすればよいでしょうか。

フォームにはエクスプローラからのドラッグ＆ドロップでファイルを受け入れるプロパティはありません。しかし、DragAcceptFilesなどのWindows APIを使って実現できます。

付録CDには、エクスプローラからのファイルのドロップを受け入れるためのFileDropコンポーネントが収録されています。

FileDropコンポーネントのControlプロパティにエクスプローラからのファイルドロップを受け入れたいコンポーネントを指定します。これは、ウィンドウハンドルを持つコンポーネントでなければなりません（LabelやSpeedButtonは指定できません）。また、空の場合は配置したフォーム自身がドロップの対象になります。FileDropコンポーネントのOnFileDropイベントでは、ドロップされたファイルの数（Num）、ファイル名リスト（Files）、位置（X, Y）が渡されます。

CHAP2¥DRGFILE.DPR（FileDropコンポーネントを使用）

Q. アプリケーションを常にタスクバーに最小化しておき、フォームを表示させないようにするには、どうすればよいでしょうか。

Applicationオブジェクトにはメインのフォームを作成しないようにするためのShowMainFormというプロパティが用意されています。プロジェクトソースでApplication.Run;を実行する前に、このプロパティにFalseを代入すればよいでしょう。

ただし、タスクバーのボタンが押されるとアプリケーションがアクティブ化されることになり、右クリックでメニューが表示しにくくなることがあります（他のアプリケーションをアクティブ化すればよい）。これに対処するためには、次のようにプログラムします。

```
type
  TForm1 = class(TForm)
    ...
    procedure FormCreate(Sender: TObject);
  protected
    procedure AppMinimize(Sender: TObject);
    ...
  end;

  ...

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnActivate := AppMinimize;
end;

procedure TForm1.AppMinimize(Sender: TObject);
begin
  Application.Minimize;
end;
```

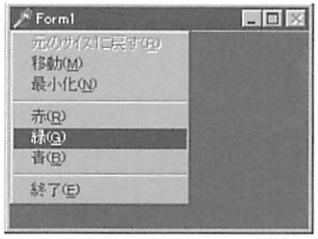
CHAP2¥KEEPMIN.DPR

Q. フォームのシステムメニューに項目を追加したいのですが、どうすればよいでしょう。

付録CDには、システムメニューにメニュー項目を追加するためのSystemMenuコンポーネントが収録されています。SystemMenuは、PopupMenuコンポーネントに似ており、ダブルクリックしてメニュー項目を追加します。しかし、PopupMenuと違い配置したフォームのシステムメニューに自分自身に設定されているメニュー項目を登録します。配置されたフォームがアプリケーションのメインフォームであれば、タスクバーに表示されるアプリケーションボタンのためのシステムメニューも変更します。

EnableCommandsを使えば、システム定義のメニュー項目を削除することもできます。ただし、いったん削除したメニュー項目を実行時に復活させることはできません。

図2-8　項目が変更されたシステムメニュー



CHAP2¥SMENU.DPR（SystemMenuコンポーネントを使用）

Q. Windows 95のトレイにアイコンを登録するにはどうすればよいでしょうか。

付録CDには、トレイにアイコンを登録するためのIconTrayコンポーネントが収録されています。IconTrayを使うと、左右のボタンに対応するポップアップメニューやマウスをクリックしたときの動作などを簡単に定義できます。IconTrayのHideMainFormをTrueにすると、メインフォームやタスクバーのアプリケーションボタンを隠すことができます。

CHAP2¥USETRAY.DPR（IconTrayコンポーネントを使用）

Q. 普通のアプリケーションは、メインウィンドウを最小化するときアニメーション（段々大きさが小さくなる）でタスクバーに格納されますが、Delphiのアプリケーションではそうなりません。

Delphiのアプリケーションでは、アプリケーションのための隠れたウィンドウが作られ、これが内部でのメインウィンドウとなっているため、Delphiとしてのメインウィンドウであるメインフォームが最小化・復元されるときにはアニメーションが禁止されます。

付録CDには、メインフォームのアニメーション動作を有効にするためのAnimateFormコンポーネントが収録されています。AnimateFormをプロジェクトのメインフォームに配置すれば、最小化または最大化する際にアニメーション動作が働くようになります。

CHAP2¥ANIMFRM.DPR

第3章

プログラミング

本章では、プログラミング言語Object Pascalに関する質問やDelphiで提供されているクラスについて取り上げています。

Q. 文の終わりに付けるセミコロン(;)の法則がわかりません。elseの前にセミコロンを付けるとエラーになりますし、endの前ではセミコロンを忘れてもエラーになりません。これは、どのように解釈すればよいでしょうか。

セミコロンは、文の「終わり」ではなく「区切り」をあらわすものと考えてください。たとえば、**if**文は**if** 条件 **then** 文1 **else** 文2;という形式をとりますが、**else**は単独の文ではありません。**else**の前にセミコロンを記述すると、そこで**if**文が終了するものとしてみなされ、**else**のところでエラーになります。

```
if 条件 then
  文1;        { 間違い }
else
  文2;
```

また、**then**や**else**の後ろに複数の文を記述するときは**begin**～**end**で囲みます。

```
if 条件 then
begin
  文1;
  文2;
  文3;
end;
```

これは、複数の文を1行に記述する場合も同様です。

```
if 条件 then begin 文1; 文2; 文3; end;
```

次のように記述した場合、条件が偽のときでも実行されないのは文1だけで、文2、文3は実行されてしまいます。

```
if 条件 then 文1; 文2; 文3;
```

begin～**end**の中に複数の文を記述する場合、**begin** 文1; 文2; 文3 **end**;のように文の間には区切りのためのセミコロンが必要ですが、**end**の直前では必要ありません。**end**の直前にセミコロンを付けてもエラーにならないのは、最後のセミコロンと**end**の間に空文（何もない文）があると解釈されているためです。

CHAP3¥IFELSE.DPR

Q. 整数型のプロパティに値を加算するために、Inc(Width, 4);のようにしているのですが、コンパイルエラーが発生してしまいます。

Incは加算を高速に実行するための組み込み関数で、第1引数には変数そのものを渡す必要があります。プロパティは、値を取得したり設定するための手段を提供するもので、単純な変数としては評価できません。プロパティに値を加算するためにはWidth := Width + 4;のように記述してください。

CHAP3¥CHGPROP.DPR

Q. "で囲んだ文字列定数でシングルクォート(')を使うにはどうすればよいでしょうか。

'を重ねて指定します。'ABC''DEF'という文字列定数は「ABC'DEF」という文字列をあらわします。

文字コードを文字として使いたい場合は#を使います。たとえば、'ABC'#68とすると'ABCD'と同じ意味になります。#を使えば、制御文字のような表記できない文字も文字列定数に埋め込めます。'ABC'#13#10は、'ABC'という文字列の後ろに復帰改行コードが追加されたものです。'ABC'#0とすれば、'ABC'という文字列の後ろにヌル文字が追加されることになります。#の後ろには16進数も使えるため、'ABC'#$44とすれば、'ABCD'と同じに意味になります。

プログラム中で、与えられた文字列からPascalとしての文字列定数を得るにはAnsiQuotedStr（またはQuotedStr）を使います。AnsiQuotedStrは、文字列の前後にシングルクォートを追加し、文字列中にシングルクォートがあれば、そこにもシングルクォートを追加します。

また、文字列がPascalの識別子として正しい形式かどうかを調べるためには、IsValidIdent関数を使います。

CHAP3¥SCONST.DPR

Q. GetActiveWindowやSetActiveWindowでアクティブなウィンドウを調べたり、設定したりしたいのですが、他のアプリケーションのウィンドウが見つけられないようです。

Win32（Windows95/NT）では、GetActiveWindowやSetActiveWindowは呼び出したアプリケーションに対してのみ有効です。また、アプリケーションが複数のスレッドを使っている場合は、対応するスレッドに対してのみ有効です。他のアプリケーションのウィンドウを対象にする場合は、GetForegroundWindowやSetForegroundWindowを使います。

CHAP3¥SETFORE.DPR

Q. Windows APIのSetFocusを呼びだそうとしているのですが、引数が間違っているというエラーになります。

SetFocusというのは、TWinControlでメソッドとして定義されているため、フォームのイベントハンドラでSetFocusを呼びだそうとすると、このメソッドの呼び出しと認識されてしまいます。こうした識別子の重複を避けるためには、ユニット名やフォーム名を明示的に指定します。

Windows APIは、Windowsというユニットで宣言されているため、Windows.SetFocus(Handle);のようにすればよいでしょう。

また、イベントハンドラなどで**with**文を使ってコンポーネントを指定しているときに、コンポーネントのプロパティ名がフォームのプロパティ名を隠してしまうような場合は、Self.Captionのように対象を明示することで、フォーム自身のプロパティにアクセスできます。

CHAP3¥WFOCUS.DPR

Q. フォーム上に、異なる目的のためにラジオボタンを配置していますが、配置する場所やタブ順序に関わらず、いずれか一つしか選べないようです。

ラジオボタンは、フォームやコンテナ上でグループ化されます。フォーム上に直接配置したラジオボタンは、どんな配置方法でもいずれか一つしか選べません。複数のラジオボタンをフォームに配置する場合は、GroupBoxやPanelなどのコンテナコンポーネント上に配置してください。また、RadioGroupコンポーネントを使うと、簡単にラジオボタン

のグループを作成できます。

図3-1は、ラジオボタンを使った配置例です。フォーム上に直接配置したラジオボタンは、位置関係に関わらずすべてひとまとまりのものとみなされます。

図3-1　ラジオボタンの配置例

CHAP3¥RBUTTONS.DPR

Q. **FormatDateTime関数を使っていますが、書式のddd（dddd）やmmm（mmmm）はいずれも日本語で曜日や月名を返します。英単語（Sunday、Januaryなど）で返す書式はないのでしょうか。**

FormatDateTimeの書式指定は、曜日や月名の表記をシステムの設定から読み込んでいます。したがって、日本語環境では日本語が使われます。たとえば、コントロールパネルの地域を選び、**[地域]**ページで「英語(U.S.)」を選ぶと、曜日の表記が英文になります(実際には、このような変更は実用的ではありません)。

これらの設定は、LongMonthNamesやShortDayNamesといったグローバル変数に読み込まれていますので、プログラム中でこれらを変更することでFormatDateTimeでの表記も変更できるようになります。

付録CDに収録されているSysCtrlsユニットには、曜日の設定を英文に設定するための手続きInitMonthDayNamesと元に戻すための手続きRestoreMonthDayNamesが含まれています。

CHAP3¥DAYNAMES.DPR（SysCtrlsユニットを使用）

Q. スクリーン全体のイメージをTBitmapオブジェクトにコピーしたいのですが、どうすればよいでしょうか。

Delphiにはスクリーン全体を指すオブジェクトは用意されていないので、Windows APIを使うことになります。次のプログラムは、スクリーン全体のイメージを持つ新しいTBitmapオブジェクトを生成する関数のプログラム例です。

```
function CreateDesktopBitmap: TBitmap;
var
  DC: HDC;
begin
  DC := GetDC(0);              {スクリーンのデバイスコンテキストを取得}
  Result := TBitmap.Create;    {ビットマップオブジェクトの生成}
  with Result do
  begin
    Width := Screen.Width;     {ビットマップの幅と高さを}
    Height := Screen.Height;   {スクリーンの大きさに合わせる}
    { Windows API を使って、イメージを転送 }
    BitBlt(Canvas.Handle, 0, 0, Width, Height, DC, 0, 0, SRCCOPY);
  end;
  ReleaseDC(0, DC);
end;
```

CHAP3¥DESKBMP.DPR

Q. Imageコンポーネントのビットマップを独自のTBitmap型変数に代入しようとしているのですが、プログラムが正常に動作しません。

TBitmapは、ビットマップを表現するための非常に強力なクラスです。ビットマップの複製や描画、ファイルとのやり取りなども自由にできます。

TBitmapなどのクラス（**class**として定義されているもの）は、すべてコンストラクタCreateを呼び出す必要があります。たとえば、Bitmap := TBitmap.Create;のようにします。

また、TBitmap型の変数は内部的にはポインタとして実装されているため、単に変数を代入するだけでは2つの変数が同じビットマップオブジェクトを指すことになります。ビットマップそのものを複製するためには、Assignというメソッドを使います。

フォーム上にImageコンポーネントがあり、ビットマップイメージが割り当てられているとき、次のプログラムはイメージを内部のビットマップオブジェクトにコピーし、輪郭を描画した上で、ファイルに出力します。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Bmp: TBitmap;
  R: TRect;
begin
  { イメージが設定されていて、ビットマップのときのみ処理する }
  if (Image1.Picture <> nil)
     and (Image1.Picture.Graphic is TBitmap) then
  begin
    { ビットマップオブジェクトの生成 }
    Bmp := TBitmap.Create;
    try
      { ビットマップの複製 }
      Bmp.Assign(Image1.Picture);
      { ビットマップへの描画 }
      with Bmp.Canvas do
      begin
        R := ClipRect;
        Brush.Style := bsClear;
        Pen.Color := clBlue;
        Pen.Width := 1;
        Rectangle(R.Left, R.Top, R.Right, R.Bottom);
      end;
      { ビットマップをファイルへ保存 }
      Bmp.SaveToFile('NEWIMAGE.BMP');
    finally
      { ビットマップオブジェクトの解放 }
      Bmp.Free;
    end;
  end;
end;
```

なお、Image.Picture.Graphic := Bmp;のようにプロパティに代入する場合は、内部で自動的にAssignメソッドを呼び出すため正しく動作します。プロパティへの代入か、単なる変数（またはクラスのフィールド）への代入かわからない場合は、Assignメソッドを使うようにしてください。

CHAP3¥BLUEFRM.DPR

Q. Imageコンポーネントにメタファイル(.EMF)を読み込むことはできますが、作成することはできないのでしょうか。

メタファイルはTMetafileというクラスで管理されていますが、このクラスには描画用のメソッドは用意されていません。メタファイルを作成するためには、Windows APIを使います。メタファイルデバイスに描画する場合にも、Canvasではなく直接デバイスコンテキストを使います。次のプログラムは、星型のメタファイルを作成するプログラム例です。

なお、Win32ではHMETAFILEの代わりにHENHMETAFILEを使うことが推奨されています。

```
procedure MakeStarMetafile;
const
  { 星型の座標 }
  StarPos: array [0..9] of Integer = (50,0,21,91,97,34,3,34,79,91);
var
  hdcMeta, hdcRef: HDC;
  R: TRect;
  hbrPrev: HBRUSH;
  hmf: HENHMETAFILE;
  mf: TMetafile;
begin
  { Windows API を使ってメモリメタファイルを作成 }
  hdcRef := GetDC(0);
  R := Rect(0, 0, 100, 100);
  hdcMeta := CreateEnhMetaFile(hdcRef, nil, @R, 'Sample Star');
  { 星型は黒で塗りつぶす }
  hbrPrev := SelectObject(hdcMeta, GetStockObject(BLACK_BRUSH));
  SetPolyfillMode(hdcMeta, WINDING);
  { 星型の描画 }
  Polygon(hdcMeta, StarPos, 5);
  SelectObject(hdcMeta, hbrPrev);
  { メモリメタファイルへのハンドルを取得 }
  hmf := CloseEnhMetaFile(hdcMeta);
  { TMetafile オブジェクトの作成 }
  mf := TMetafile.Create;
  mf.Handle := hmf;                    { 最初にハンドルを代入する }
  mf.SaveToFile('C:¥WORK¥STAR.EMF');{ ファイルに保存 }
  mf.Free;                             { オブジェクトの解放 }
  ReleaseDC(0, hdcRef);
end;
```

CHAP3¥MKMETAS.DPR

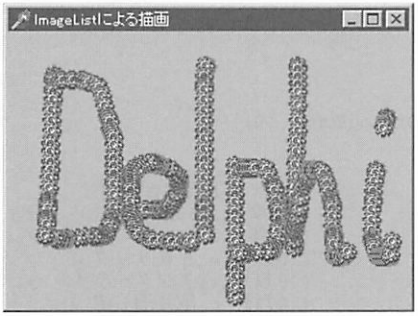
Q. ビットマップの一部を透明にしたいのですが、どうすればよいでしょうか。

ビットマップに透明という設定はありません。しかし、ImageListを使うと特定の色を透明をあらわす代わりに使えます。

ImageListにビットマップを登録するとき、通常はビットマップの左下の色が**[透過色(T)]**として使われます。もし、透明にしたい色がなければ最後のclNoneを選びます。透過色は、イメージを追加するときしか設定できません。また、あらかじめHeight、Widthプロパティでイメージの大きさを指定しておく必要があります。

ImageListに登録したビットマップを描画する場合は、Drawメソッドを使います。たとえば、フォームのOnMouseDownイベントハンドラでImageList1.Draw(Canvas, X, Y, 0);とすれば、マウスをクリックした場所にImageListの先頭のイメージが描画されます。

図3-2　ImageListを使った描画



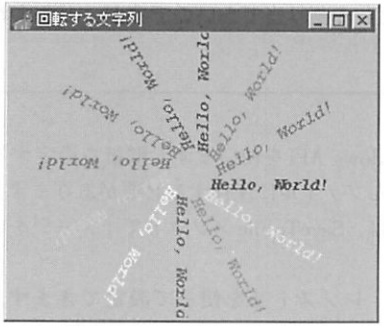
CHAP3¥DRAWIMG.DPR

Q. 文字列を斜めに描画することはできますか。

コンポーネントやCanvasなどで使われているFontプロパティでは、文字の描画方向を指定することはできません。このため、Windows APIを使って方向を指定する必要があります。次のプログラムは、フォーム上にPaintBoxコンポーネントを配置し、OnPaintイベントハンドラを定義してさまざまな角度で文字列を描画するものです。

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
const
  ColorTable: array [0..11] of TColor =
      (clBlack, clGreen, clOlive, clNavy, clPurple, clTeal,
       clRed, clLime, clYellow, clBlue, clFuchsia, clAqua);
var
  Angle: Integer;
  Index: Integer;
  LogFont: TLogFont;
begin
  Angle := 0;
  Index := 0;
  while Angle < 3600 do
  begin
    FillChar(LogFont, SizeOf(LogFont), 0);
    with LogFont do
    begin
      lfHeight := 16;              { フォントの大きさ }
      lfEscapement := Angle;       { 角度(0..3600) = (0..2π) }
      lfWeight := FW_BOLD;         { 書体 }
      lfItalic := 1;               { 0以外 = イタリック }
      lfUnderline := 0;            { 0以外 = アンダーライン }
      lfStrikeOut := 0;            { 0以外 = 取り消し線 }
      StrPCopy(lfFaceName, 'Courier New');  { フォント名 }
    end;
    with PaintBox1, Canvas do
    begin
      { Windows API を使ってフォントを作成する }
      Font.Handle := CreateFontIndirect(LogFont);
      Font.Color := ColorTable[Index];
      Brush.Style := bsClear;
      { 文字列を描画する }
      TextOut(Width div 2, Height div 2, '  Hello, World!');
    end;
    Inc(Angle, 300);
    Inc(Index);
  end;
end;
```

図3-3　文字列をさまざまな角度で描画



作成したフォントは、不要になった時点で自動的に削除されます。

CHAP3¥ANGLESTR.DPR

Q. プログラムで作成したイメージを壁紙として割り当てたいのですが、どうすればよいでしょうか。

SystemParametersInfoというWindows APIを使います。壁紙を設定するためには、プログラム中のビットマップをいったんファイルに保存する必要があります。プログラム中でTBitmapクラスを使っている場合は、SaveToFileメソッドでイメージをファイルに保存できます。

壁紙のファイル名や表示形式は、レジストリを使って設定できます。具体的には、HKEY_USERS¥Control Panel¥desktopキーのWallpaperエントリとTileWallpaperエントリを使います。

設定した内容を反映させるためには、SystemParametersInfoにSPI_SETDESKWALLPAPERを渡します。壁紙のファイル名だけを変更したい場合は、SPI_SETDESKWALLPAPERだけを使うこともできます。

次の手続きは、指定したファイル名と表示スタイルを設定します。

```
uses Registry;

procedure SetWallPaper(AFile: TFileName; ATile: Boolean);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_USERS;
  if Reg.OpenKey('.Default¥Control Panel¥desktop', False) then
  begin
    Reg.WriteString('Wallpaper', AFile);
    if ATile then
      Reg.WriteString('TileWallpaper', '1')
    else
      Reg.WriteString('TileWallpaper', '0');
  end;
  SystemParametersInfo(SPI_SETDESKWALLPAPER, 0, nil, 0);
  Reg.Free;
end;
```

Windows APIのSystemParametersInfoは、壁紙だけでなくコントロールパネルで設定する他の項目を変更したり参照するためにも使えます。たとえば、SPI_GETNONCLIENTMETRICSを使うとウィンドウのキャプションやメニューのような非クライアント領域で使われている文字フォントの情報などを取得できます。

CHAP3¥WEEKWALL.DPR（ShellLinkコンポーネントを使用）

Q. Delphiで作成するアプリケーションから、他のプログラムを呼び出したいのですが、どうすればよいでしょうか。

単純にプログラムを呼び出すだけならば、WinExecというWindows APIを使うのがもっとも簡単です。WinExecは第1引数にプログラム名を、第2引数に表示状態を指定します。たとえば、CALC.EXE(電卓)を呼び出す場合は、次のように記述します。

```
WinExec('CALC.EXE', SW_SHOW);
```

指定されたプログラムは次の順序で検索されます。

1. 実行されるアプリケーションのあるディレクトリ
2. カレントディレクトリ
3. Windowsのシステムディレクトリ（NTの場合は、SYSTEM32／SYSTEMの両方）
4. Windowsディレクトリ
5. 環境変数PATHで指定されているディレクトリ

これらに該当しないディレクトリにあるプログラムは、パス名付きで呼び出します。

```
WinExec('C:\WINDOWS\REGEDIT.EXE', SW_SHOW);
```

WinExecは、プログラムの呼び出しに失敗すると32未満の値を返します。

また、Win32ではCreateProcessというWindows APIを使うことが推奨されています。CreateProcessは、詳細な指定が必要な代わりに実行ファイル名、完全なコマンドライン、セキュリティ情報などを指定して新しいプログラムを呼び出せます。次のプログラムは、自分自身のフォームのクライアント領域にメモ帳（NOTEPAD.EXE）を表示させるプログラムです。

```
var
  StartupInfo: TStartupInfo;
  ProcessInfo: TProcessInformation;
  WinDir: array [0..255] of Char;
  CmdFile, CmdLine: string;
begin
  GetWindowsDirectory(WinDir, SizeOf(WinDir));
  { 特に指定が必要ない場合は、GetStartupInfo を使う }
  { GetStartupInfo(StartupInfo); }
  with StartupInfo do
  begin
    cb := SizeOf(StartupInfo);
    lpReserved := nil;
```

```
    lpDesktop := nil;
    lpTitle := nil;
    dwX := ClientOrigin.X;       { クライアント領域を }
    dwY := ClientOrigin.Y;       { 実行領域として使う }
    dwXSize := ClientWidth;
    dwYSize := ClientHeight;
    dwXCountChars := 0;
    dwYCountChars := 0;
    dwFillAttribute := 0;
    dwFlags := STARTF_USESIZE or STARTF_USEPOSITION;
    wShowWindow := SW_SHOW;
    cbReserved2 := 0;
    lpReserved2 := nil;
    hStdInput := 0;
    hStdOutput := 0;
    hStdError := 0;
  end;
  CmdFile := StrPas(WinDir) + '¥NOTEPAD.EXE';
  CmdLine := CmdFile + ' README.TXT';
  CreateProcess(PChar(CmdFile), PChar(CmdLine), nil, nil,
  False, 0, nil, WinDir, StartupInfo, ProcessInfo);
```

CreateProcessを使うことで、生成されたプロセスのハンドルを調べ、プロセスの終了を待つことができます（Win32では、Windows 3.1で使われていたGetModuleUsageは使えません）。上記のプログラムに以下の記述を追加することで、呼び出したメモ帳が終了するまで待ちます。

```
  while WaitForSingleObject(ProcessInfo.hProcess, 0) = WAIT_TIMEOUT do
    Application.ProcessMessages;
```

この他の便利なWindows APIとしてShellExecuteがあります。これは、ShellAPIというユニットで定義されているもので、WinExecと同じように実行ファイルを呼び出すだけでなく、起動ディレクトリを指定したり、Windowsの関連付けを利用してアプリケーションを起動することができます。

ShellExecuteを使う例を以下に示します。

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ShellAPI;   { ShellAPI を追加する }

  ...

procedure TForm1.Button1Click(Sender: TObject);
begin
  { カレントディレクトリを C:¥Program Files¥Borland¥Delphi 3 にし、}
  { メモ帳(NOTEPAD.EXE)を起動する }
  ShellExecute(Handle, 'OPEN', 'NOTEPAD.EXE', '',
               'C:¥Program Files¥Borland¥Delphi 3', SW_SHOW);
end;
```

たとえば、次のようにすれば.BMPという拡張子に関連付けられている「ペイントブラシ」が自動的に呼び出されて、花見.BMPがオープンされます。

```
{ C:¥Windows¥花見.BMP を直接指定する }
ShellExecute(Handle, 'OPEN', 'C:¥Windows¥花見.BMP', '', '', SW_SHOW);
```

フォームにButtonと3つのEditコンポーネントを配置して、Button1のイベントハンドラでプログラム名、コマンドライン、起動ディレクトリを指定して実行するには以下のようにします。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Windows API に文字列を渡す場合は、PChar でキャストする }
  ShellExecute(Handle, 'OPEN', PChar(Edit1.Text), PChar(Edit2.Text),
               PChar(Edit3.Text), SW_SHOW);
end;
```

CHAP3¥EXECCMD.DPR

Q. **アプリケーションから直接Windows95を再起動させたいのですが、どうすればよいでしょうか。**

Windows APIのExitWindowsExを使います。ExitWindowsExの第1引数に与える引数と意味は、以下の組み合わせとなります。第2引数は将来のために予約されており、使われません。システムを再起動させるためには、ExitWindowsEx(ewx_Reboot, 0);となります。

なお、Windows 3.1で使われていたExitWindowsは引数に関わらずExitWindowsEx(ewx_Logoff, -1);と同じ動作になります。

ewx_Force	アプリケーションの応答を待たず、システムを強制的に終了させる
ewx_Logoff	システムをシャットダウンし、ログオフする
ewx_Reboot	システムをシャットダウンし、再起動する
ewx_PowerOff	システムをシャットダウンし、電源を切る
ewx_ShutDown	システムをシャットダウンし、電源を切っても安全な状態にする

CHAP3¥EXITWIN.DPR

Q. **Printerオブジェクトを使って、プリンタへ出力するときに、PrinterSetupDialogを使わずに直接印字方向（縦、横）を切り換えることはできませんか。**

Printerオブジェクトには、実行時に使えるOrientationというプロパティがあります。このプロパティにpoPortraitを代入すれば縦方向に、poLandscapeを代入すれば、横方向に印字できるようになります。

CHAP3¥PRTEST.DPR

Q. **プリンタに印字する際、フォントや大きさはどのように設定すればよいのでしょうか。**

PrinterオブジェクトのCanvasプロパティがプリンタ用のデバイスコンテキストをあらわしています。そして、Canvas.Fontプロパティがプリンタに出力するフォントを表わしています。たとえば、Canvas.Font.Name := 'Arial'; Canvas.Font.Size := 20;とすれば、Arialの20ptフォントが使われます。

ただし、新しいフォントを割り当てるとフォントの情報がプリンタの解像度ではなく、画面の解像度になってしまうことがあります。この場合は、Printer.Canvas.Font.PixelsPerInch := GetDeviceCaps(Printer.Handle, LOGPIXELSY);としてください。

CHAP3¥PRTEST.DPR

Q. **プリンタに印字する際、特定の用紙トレイを使って印刷したいのですがどうすればよいでしょうか。**

Windows 95の[スタート]メニューから[設定｜プリンタ(P)]を選び、使いたいプリンタのプロパティの[用紙]ページで「給紙方法」を設定しておけば、そのプリンタを使う際のデフォルトの用紙トレイを決められます。

デフォルトの設定を変更せず、アプリケーションだけで特定の用紙トレイを使いたい場合は、プリンタデバイスの情報を持つTDeviceModeレコードを直接扱う必要があります。現在選択されているプリンタやデバイスの状態はPrinterオブジェクト（TPrinter型）のGetPrinterメソッドを使って得られます。用紙トレイは、dmDefaultSourceフィールドが

示します。

しかし、用紙トレイの種類はプリンタごとに違うため、統一した値では設定できません。このため、一度はPrinterSetupDialogなどを使ってプリンタの状態を設定しておき、設定された値を次回以降で利用します。

Win32オンラインヘルプでは、dmDefaultSourceは予約領域となっており、常に0を指定することになっています。

以下の関数や手続きを使えば、現在の給紙トレイの設定を参照したり、設定することができます。ただし、プリンタが切り換えられるとトレイを示す値は無効になりますので、注意してください。

付録CDのサンプルでは、アプリケーションが終了するときに、設定されている用紙トレイをレジストリに記録し、2回目以降の起動時に用紙トレイの情報をプリンタの設定に反映させています。

```
uses Printers;

function GetCurrentPrinterTray: Word;
var
  DeviceMode: THandle;
  DevMode: PDeviceMode;
  Device, Driver, Port: array [0..79] of Char;
begin
  Printer.GetPrinter(Device, Driver, Port, DeviceMode);
  if DeviceMode <> 0 then
  begin
    DevMode := GlobalLock(DeviceMode);
    Result := DevMode^.dmDefaultSource;
    GlobalUnlock(DeviceMode);
  end;
end;

procedure SetCurrentPrinterTray(Value: Word);
var
  DeviceMode: THandle;
  DevMode: PDeviceMode;
  Device, Driver, Port: array [0..79] of Char;
begin
  Printer.GetPrinter(Device, Driver, Port, DeviceMode);
  if DeviceMode <> 0 then
  begin
    DevMode := GlobalLock(DeviceMode);
    DevMode^.dmDefaultSource := Value;
    GlobalUnlock(DeviceMode);
  end;
end;
```

CHAP3¥PRTEST.DPR

Q. PrinterオブジェクトにImageコンポーネントの内容（Image1.Picture.Graphic）を出力するため、CanvasプロパティのDrawメソッドを使いましたが、プリンタには何も出力されません。どうすれば出力できるようになるでしょうか。

Drawメソッドでは、ビットマップやアイコンを簡単にCanvasに描画できます。しかし、モノクロプリンタなどディスプレイのデバイスコンテキストと互換性のない場合は、正常にビットマップイメージを転送できないことがあります。

ディスプレイデバイスとプリンタデバイスで互換性のない場合にも、正しくビットマップイメージを出力するために、いったんDIB(Device Independent Bitmap)を作成して出力する手続きを以下に示します。

```
procedure PrintBitmap(X, Y: Integer; Bitmap: TBitmap);
var
  Info: PBitmapInfo;
  InfoSize: Integer;
  Image: Pointer;
  ImageSize: DWord;
begin
  with Bitmap do
  begin
    GetDIBSizes(Handle, InfoSize, ImageSize);
    Info := AllocMem(InfoSize);
    try
      Image := AllocMem(ImageSize);
      try
        GetDIB(Handle, Palette, Info^, Image^);
        with Info^.bmiHeader do
          StretchDIBits(Printer.Canvas.Handle, X, Y, Width, Height,
            0, 0, biWidth, biHeight, Image, Info^, DIB_RGB_COLORS,
            SRCCOPY);
      finally
        FreeMem(Image, ImageSize);
      end;
    finally
      FreeMem(Info, InfoSize);
    end;
  end;
end;
```

なお、プリンタデバイスによってはStretchDIBitsをサポートしていないものもあります。デバイスがある機能をサポートしているかどうかは、GetDeviceCapsというWindows APIを使って調べることができます。たとえば、「(GetDeviceCaps(Printer.Canvas.Handle, RASTERCAPS) **and** RCSTRETCHDIB) <> 0」とすれば、StretchDIBitsがサポートされている場合にはTrue（真）に、サポートされていなければFalse（偽）になります。

CHAP3¥PRTEST.DPR

Q. 印刷のプレビュー画面を作りたいのですが、よい方法はないでしょうか。

Delphi 3のレポート出力コンポーネントQuickReportを使っている場合は、QuickRepのPreviewメソッドを呼び出すだけで、プレビュー画面が表示されます。また、QRPreviewコンポーネントを使えば独自の範囲をプレビュー領域として使えます（OnPreviewイベントハンドラの記述でTQRPrinterが未定義エラーになる場合は、**uses**節にQRPrntrを追加してください）。

Printerオブジェクトを使って、直接プログラムでプリンタを制御する場合は、通常のPrinter.Canvasと同様にフォームやPaintBoxのCanvasに描画することで、プリンタへの印字と同じ内容を描画できます。このとき、プリンタと画面上ではインチあたりのドット数（PixelsPerInch）が違うこと、イメージの出力ではDrawメソッドが使えない場合があること（前項目参照）などの注意が必要です。たとえば、フォームの解像度が96dpi（インチあたりの96ドット）の場合は、1センチの幅をあらわすために、約38ドット必要です（1.0×96/2.54=約37.8）。

付録CDのサンプルプログラムでは、拡大・縮小にも対応したプレビュー機能を実現しています（図3-4）。

図3-4　サンプルプログラムでのプレビュー

CHAP3¥PRTEST.DPR

Q. テキストをプリンタに出力したいだけなのですが、簡単な方法はありませんか。

テキストファイル変数を定義して、PrintersユニットのAssignPrn手続きを使ってプリンタデバイスを割り当てることができます。簡単な例を以下に示します。

```
var
  PrnOut: TextFile;
begin
  AssignPrn(PrnOut);
  Rewrite(PrnOut);
  Writeln(PrnOut, 'これはプリンタに出力するサンプルです。');
  CloseFile(PrnOut);
end;
```

CHAP3¥PRTEXT.DPR

Q. Delphi 1.0で、WritelnやReadlnを使うためにWinCrtを使っていたのですが、Delphi 3にはWinCrtはないのでしょうか。

Delphi 3にはWinCrtユニットはありませんが、代わりに32ビットのコンソール画面を利用できます。**[プロジェクト(P)｜オプション(O)]**の**[リンカ]**ページで、**[コンソールアプリケーションの作成(C)]**をチェックすれば、WinCrtを使う場合と同様、WritelnやReadlnを使えます。この入出力は、MS-DOSプロンプトが対象となります。いっさいフォームを使わない場合は、すべてのフォームをプロジェクトから削除しプロジェクトソースを次のように書換えます。このプログラムは、VCLを使わないため非常に小さい実行ファイルができあがります。

```
program CONAPP;

uses Windows;    { プロジェクトの形式として必要な行 }

begin
  Writeln('Hello, Delphi programmers! [press Enter]');
  Readln;        { [ENTER] を押した時点で終了させる }
end.
```

CHAP3¥CONAPP.DPR

Q. C言語のoutp/inpのように、Object Pascalで直接I/Oポートを制御することはできますか。Delphi 1.0で使っていたPort配列は使えないようです。

Windowsアプリケーションが直接ハードウェアを操作することはあまり好ましくありません。Delphi 1.0では、従来のTurbo Pascalからの仕様としてPort変数というものが使えるようになっていましたが、これはDelphi 3では使えません。しかし、次のようにインラインアセンブリを使うことで直接I/Oポートを制御できます。

```
procedure TForm1.Button1Click(Sender: TObject);
asm
   { DOS/V でビープ音を鳴らす }
   mov dx, $0061
   in  al, dx
   or  al, $03
   out dx, al
end;

procedure TForm1.Button1Click(Sender: TObject);
asm
   { DOS/V でビープ音を止める }
   mov dx, $0061
   in  al, dx
   and al, $FC
   out dx, al
end;
```

付録CDに収録されているSysCtrlsユニットには、直接I/Oポートを制御できるクラスが含まれています。これはI/Oポートを配列のようにみなして使うもので、配列要素への代入が出力(out)、配列要素の参照が入力(**in**)として機能します。たとえば、PC-9800シリーズではPort[$37] := 6;とすればビープ音が鳴り、Port[$37] := 7;とすればビープ音が止まります。もちろん、こうしたハードウェアの仕様に依存するプログラムは、互換性のない他の機種では実行できなくなります。

使用中の機種がDOS/Vであるか PC-9800であるかを調べる方法のひとつとして、Windows APIのGetKeyboardTypeを使うことができます。GetKeyboardTypeの引数に1（サブタイプの取得）を指定すると、PC-9800の場合には$0D01～$0D04が、DOS/Vの場合は$0000～$0004という値が返されます。このため、返された値の上位バイトが$0Dであるかどうかでで PC-9800か DOS/Vかを区別できます。

CHAP3¥IOPORT.DPR（SysCtrlsユニットを使用）

Q. Cのような文字判定ルーチンはないのでしょうか。

標準ライブラリには文字判定ルーチンはありませんが、同様の関数を作成することはできます。付録CDに収録されているSysCtrlsユニットでは、Borland C++と互換性を持つ文字列判定関数を実現する関数群が提供されています。

これらの関数を使いたい場合は、**uses**節にSysCtrlsを追加します。

```
implementation

{$R *.DFM}

uses SysCtrls;

procedure TForm1.Button1Click(Sender: TObject);
var
  c: Char;
begin
   ...
   if isalnum(c) then ...
```

CHAP3¥CHKCHAR.DPR（SysCtrlsユニットを使用）

Q. Delphiでコンポーネントを作成していますが、アクセス制御を使って上位クラスに指定したメンバを下位クラスで隠すにはどうすればよいでしょうか。C++では、継承するときに**private**や**protected**で宣言しなおせます。

DelphiのObject Pascalでは、C++のアクセス制御とは考え方が異なります。つまり、継承するときにアクセス制御を緩めることはできますが、制約することはできません。このため、上位クラスに指定したコンポーネントで**public**や**published**に指定されたプロパティは、すべて新しいクラスでも参照できることになります。

なお、Delphiのビジュアルコンポーネントライブラリには、TLabelやTEditとは別にTCustomLabelやTCustomEditなどの拡張専用コンポーネントが定義されています。これらは、ほとんどのプロパティが**private**や**protected**で定義されているため、継承するときに好きなプロパティだけを**public**や**published**で宣言しなおすことができます。ただし、TCustomGridのように、拡張したクラスでメンバを再定義しなければ使えないものもあります。

Q. Delphiのプログラムでコールバック関数は使えますか。

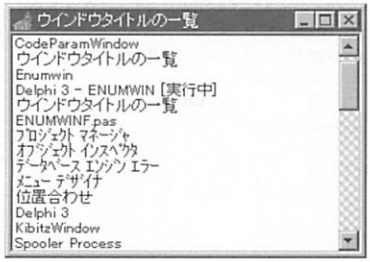
DelphiのObject Pascalでは、ウィンドウハンドルやコールバック関数が使えます。16ビットアプリケーションでは、スマートコールバックを使うか、MakeProcInstance/FreeProcInstanceといったWindows APIを呼び出す必要がありましたが、Win32では不要です。

Win32では、一般的なコールバック関数は**stdcall**呼び出し形式を使います。16ビットアプリケーションではPASCAL形式が使われるため**export**以外の指定は必要ありませんでしたが、32ビットアプリケーションでは**stdcall**を指定する必要があります（逆に**export**は不要です）。

新規フォームにListBoxコンポーネントを貼り付け、フォームのOnCreateイベントハンドラとコールバック関数を次のように定義します。このプログラムは、Windows上でタイトルを持つすべてのウィンドウをリストボックスに表示します。

```
{ EnumWindows で使うコールバック関数(エクスポート関数) }
function EnumTitlesProc(Handle: HWnd; Info: Pointer): Bool; stdcall;
var
  title: array [0..255] of Char;     { タイトルを受け取るバッファ }
begin
  GetWindowText(Handle, title, 255); { ウィンドウタイトルの取得 }
  if StrLen(title) <> 0 then
    Form1.ListBox1.Items.Add(StrPas(title)); { ListBox1 に追加 }
  Result := True;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  EnumWindows(@EnumTitlesProc, 0);   { コールバック関数を使った例 }
end;
```

図3-5　ウィンドウタイトルの一覧表示



CHAP3¥ENUMWIN.DPR

Q. Object Pascalでファイルをアクセスする方法がわかりません。

Object Pascalで扱える基本的なファイルとして、型なしファイル、型つきファイル、テキストファイルがあります。また、より高度なファイル処理のためにTFileStreamというクラスが定義されています。

ここでは、基本的なファイル操作について説明します。Object Pascalではファイルを操作するためにはファイル変数を使います。ファイル変数を定義するときに指定する型でファイルの種類が決まります。型なしファイルはFile、型つきファイルはFile **of** 型名、テキストファイルはTextFileという型名を使います。

型なしファイルを使ったプログラム例を以下に示します。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  F: file;                          { 型なしファイル変数 }
  Buffer: array [0..7] of Char;     { 読み込みバッファ }
  RecNum: Word;                     { 読み込みレコード数 }
begin
  { OpenDialog コンポーネントを配置しておく }
  if OpenDialog1.Execute then    { 「ファイルを開く」ダイアログ }
  begin
    AssignFile(F, OpenDialog1.FileName); { ファイル名の割り当て }
    Reset(F, 8);                         { レコードサイズは8バイト }
    BlockRead(F, Buffer, 1, RecNum);     { 1レコード読み込む }
    { 読み込んだ結果でファイルの情報を表示 }
    if RecNum < 1 then
      ShowMessage('Unknown file')
    else if StrLComp(Buffer, 'MZ', 2) = 0 then
      ShowMessage('Executable file')
    else if StrLComp(Buffer, 'BM', 2) = 0 then
      ShowMessage('Bitmap file')
    else if StrLComp(Buffer, 'DCU1', 4) = 0 then
      ShowMessage('Delphi 1.0 Unit object')
    else if StrLComp(Buffer, 'HSPP', 4) = 0 then
      ShowMessage('Delphi 2.0 Unit object')
    else if StrLComp(Buffer, #$41#$86#$51#$44, 4) = 0 then
      ShowMessage('Delphi 3 Unit object')
    else if StrLComp(Buffer, 'program', 7) = 0 then
      ShowMessage('Delphi project source')
    else if StrLComp(Buffer, 'unit', 4) = 0 then
      ShowMessage('Delphi unit source')
    else
      ShowMessage('Unknown file');
    CloseFile(F);
  end;
end;
```

AssignFile手続きは、ファイル変数に実際のファイル名を割り当てるために必ず必要です。ただし、ファイル名を割り当てるだけでオープンするわけではありません。ファイルをオープンするには、ResetとRewriteという手続きを使い、それぞれ既存のファイルをオープンするか、新規にファイルを作成するかという違いがあります。ここでは既存のファイルから情報を読み込むために、Reset手続きを使っています。

Resetを使う場合、デフォルトではファイルを「読み書き用」にオープンすることに注意してください。読み込み専用でオープンするためには、Resetを呼び出す前にFileMode変数に0を代入しておきます。ただし、テキストファイルに対してはResetは読み込み専用でオープンします。

Resetにはファイル変数に加えてレコードサイズを指定できます。レコードサイズは、型なしファイルの場合だけに指定できるもので、データを読み書きするときの単位とバイト数であらわします。型なしファイルからデータを読み書きするには、BlockRead/BlockWriteという手続きを使います。レコードサイズを1にしておくとバイト単位でデータを読み書きできるようになります。

ファイルを使い終わったら、CloseFile手続きでファイルを閉じます。CloseFileする前に、再度ResetやRewriteでファイルをオープンできます。ただし、ファイルの位置は先頭に戻されます。ファイルの途中からレコードサイズを変更することはできません。

型つきファイルは、**var** F: File **of** Longint;のように定義します。型には基本的な型以外にレコード型なども利用できます。型つきファイルでも、AssignFile、Reset/Rewrite、CloseFileの意味は同じです。ただし、型つきファイルに対してデータを読み書きするには、Read/Write手続きを使います。

型なしファイルや型つきファイルではSeekを使って任意の位置のデータを読み書きできます。Seekはレコードサイズや型の大きさを元にしてファイルの位置を指定します(バイト数ではありません)。現在の場所を調べるなら、FilePos関数を使います。

テキストファイルは、**var** F: TextFile;のように定義します。型つきファイルでは、Reset/Rewriteはそれぞれ読み込み専用、書き込み専用となります。データの読み書きのためにはRead/Writeを使いますが、出力するデータはすべてテキスト文字列として書き込まれます。また、SeekやFilePosは使えません。

たとえば、AUTOEXEC.BATを1行ずつ読み込むためには次のようにプログラムします。

```
var
  { イベントハンドラの呼び出しごとに初期化しないため }
  { イベントハンドラの外で変数を定義する }
  AssignedFlag: Boolean; { ファイルをオープンしているかどうか }
  F: TextFile;           { テキストファイル変数 }

procedure TForm1.Button5Click(Sender: TObject);
var
  s: string;
begin
  if not AssignedFlag then
  begin
    { ファイルを割り当てていなければ、ファイルを割り当てる }
```

```
    AssignFile(F, 'C:¥AUTOEXEC.BAT');
    Reset(F);
    AssignedFlag := True;
  end;
  if Eof(F) then
  begin
    { ファイルが終わりになったら、ファイルを閉じる }
    CloseFile(F);
    AssignedFlag := False;
  end
  else
  begin
    { ファイルからテキストを1行読み込む }
    Readln(F, s);
    { 読み込んだテキストをメモに追加する }
    Memo1.Liens.Add(S);
  end;
end;
```

ファイル操作に関する手続きや関数については、オンラインヘルプの「入出力/標準手続き」を参照してください。型なしファイルについては「型なしファイル」、テキストファイルについては「テキストファイル」を参照してください。

TFileStreamクラスの使用例については、オンラインヘルプの「TFileStream」「TStream」や第6章「Visual Basic」のQ&Aを参照してください。また、より低レベルの入出力を行なうために、FileOpen/FileCloseなどの関数があります。

CHAP3¥IDENTFIL.DPR
CHAP3¥VIEWINI.DPR

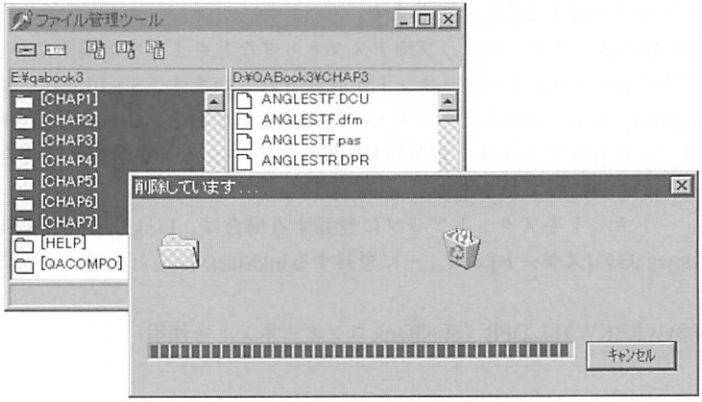
Q. ファイルを完全に削除する代わりにごみ箱に入れたり、ディレクトリごと削除するにはどうすればよいでしょうか。

Windows APIのDeleteFileは、ひとつのファイルを完全に削除します。このため、エクスプローラで削除する場合のように「ごみ箱」を使ってファイルを復活させることができません。

付録CDには、ShellAPIを使ってファイル操作をするためのFileOperationコンポーネントが収録されています。FileOperationでは、FromFilesプロパティにコピーや移動の元になるディレクトリまたはファイル名を登録し、ToFilesプロパティにコピー先のディレクトリ名を登録して準備します。実際の処理には、CopyFiles、DeleteFiles、RemoveFiles、RenameFilesメソッドを使います。単一のファイルやディレクトリを対象とするときは、CopyFile、DeleteFile、RemoveFile、RenameFileメソッドを使います。

デフォルトでは、OptionsプロパティのfofAllowUndoが有効になっているため、ごみ箱を使ってファイルを復元できます。fofAllowUndoをFalseにすれば、ファイルは直ちに削除されます。

図3-6　FileOperationコンポーネントの使用例



CHAP3¥FILER.DPR（FileOperationコンポーネントを使用）

Q. 新しいディレクトリを作成する際に、深い階層のディレクトリを一度に作成することはできませんか。

通常、CreateDirectoryのようなWindows APIを使ってディレクトリを作成する場合、複数の階層にわたるディレクトリでは1段ごとに作成しなければなりません。こうしたディレクトリを一度に作成するために、DelphiではForceDirectoriesという手続きが用意されています。たとえば、ForceDirectories('C:¥TESTAPPS¥SUB¥INFO');とすれば、たとえC:¥TESTAPPSディレクトリがなくても、一度にディレクトリを作成できます。

Q. Windows 95を起動したときに、アプリケーションを自動的に起動させたいのですが、どうすればよいでしょうか。

Windows 95の[スタート]メニューから[設定(S)｜タスクバー]を選び、[スタートメニューの設定]ページで[詳細]ボタンを押し、[プログラム｜スタートアップ]項目に、プログラムを追加します。しかし、アプリケーションの利用者にこうした手間をかけさせないため、プログラムで登録することもできます。

付録CDには、スタートアップやデスクトップなどさまざまな場所にショートカットを登録できるShellLinkコンポーネントが収録されています。ShellLinkでは、FileNameやDescription、Argumentsなどのプロパティを設定しておき、CreateLinkを呼び出してショートカットを作成できます。自分自身のアプリケーションを登録する場合は、FileNameプロパティを空にしておきます。

ショートカットをスタートアップに登録する場合は、LinkTypeをlnkStartupにします(Windows 95の[スタート]メニューに登録するlnkStartMenuと間違えないでください)。

CHAP3¥WEEKWALL.DPR (ShellLinkコンポーネントを使用)

Q. Windows 95の長いファイル名から短いファイル名を調べたり、短いファイル名から長いファイル名を調べるにはどうすればよいでしょうか。

Windows 95では、DOSスタイルの8.3形式以上の長いファイル名を扱えます。この長いファイル名を扱う場合でも、常に旧式の短いファイル名が割り当てられます。

しかし、ある特定の長い名前がどのような短い名前に割り当てられるかは、状況によって異なります。たとえば、すでにLONGFI~1という短いファイル名があれば、新しくLONGFILENAMEというファイルを作成するときの対応する短いファイル名はLONGFI~2になります。つまり、長いファイル名と短いファイル名の対応を調べるには、実際にディスク上のファイルを調べてみなければなりません。

次のプログラムは、与えられたファイル名（パス指定付き）からディスク上のファイルを調べ、それぞれ短いファイル名と長いファイル名を返す関数です。たとえば、GetShortFileNameに短いファイル名を与えてもかまいません（同じファイル名が返されます）。

```
function GetShortFileName(AFileName: TFileName): TFileName;
var
  SRec: TSearchRec;
begin
  if FindFirst(AFileName, faAnyFile, SRec) = 0 then
    Result := SRec.FindData.cAlternateFileName
  else
    Result := '';
  FindClose(SRec);
end;

function GetLongFileName(AFileName: TFileName): TFileName;
var
  SRec: TSearchRec;
begin
  if FindFirst(AFileName, faAnyFile, SRec) = 0 then
    Result := SRec.FindData.cFileNAme
  else
    Result := '';
  FindClose(SRec);
end;
```

Q. N88-BASICなどで作成したデータファイルをC++やDelphiで読み込んで使いたいのですが、整数は正しく読み込めるのに、浮動小数値は正常な値になりません。

Delphiは、浮動小数値を表現するための内部表現にIEEE形式を使っています。これに対して、PC-9800用のN88-BASICやIBM-PC用のGW-BASICでは、マイクロソフトバイナリ(MSBIN)形式が使われています。両者の形式が異なっているため、単純にデータを読み込むだけでは正常な値として利用することはできません。

以下のプログラムは、MSBIN形式の値をIEEE形式に変換します。

```
{ MSBIN 形式の倍精度実数を IEEE 形式に変換 }
procedure dmsbintoieee(var Value: Double);
var
  SrcLo, SrcHi: Longint;
  DstLo, DstHi: Longint;

begin
  SrcLo := PLongint(@Value)^;
  SrcHi := PLongint(Longint(@Value) + 4)^;
  DstHi := (((SrcHi shr 24) and $FF) - 129 + 1023) shl 20;
  if (SrcHi and $800000) <> 0 then
    DstHi := DstHi or $80000000;
  DstHi := DstHi or ((SrcHi shr 3) and $000FFFFF);
  DstLo := (SrcHi shl 29) or (SrcLo shr 3);
  PLongint(@Value)^ := DstLo;
  PLongint(Longint(@Value) + 4)^ := DstHi;
end;

{ MSBIN 形式の単精度実数を IEEE 形式に変換 }
procedure fmsbintoieee(var Value: Single);
var
  Src, Dst: Longint;
begin
  Src := PLongint(@Value)^;
  Dst := (Src shr 24) and $000000FF;
  if Dst < 2 then
  begin
    Value := 0;
    Exit;
  end;
  Dst := (Dst - 2) shl 23;
  if (Src and $00800000) <> 0 then
    Dst := Dst or $80000000;
  Dst := Dst or (Src and $007FFFFF);
  PLongint(@Value)^ := Dst;
end;
```

CHAP3¥CVMSBIN.DPR

Q. **Delphi 1.0でWindows APIを呼び出すために、文字列の先頭のアドレスを渡していたのですが、Delphi 3では正しく動作しないことがあります。**

Delphi 1.0の文字列型は、特に指定しない限り255文字の領域を確保していました。Delphi 3の文字列型は、必要に応じて文字列領域が確保されるため、何も文字列を代入していない場合には領域が確保されていません。たとえば、GetWindowsDirectoryのように文字列をバッファとして使いたい場合、次のようにあらかじめ文字列長を指定しておく必要があります。

```
var
  dir: string;
begin
  SetLength(dir, 255);
  GetWindowsDirectory(PChar(dir), 255);
  SetLength(dir, StrLen(PChar(dir)));
  Caption := dir;
end;
```

次のように文字型の配列としてプログラムすることもできます。

```
var
  dir: array [0..255] of Char;
begin
  GetWindowsDirectory(dir, 255);
  Caption := StrPas(dir);
end;
```

CHAP3¥WINDIR.DPR

第4章

コンポーネント

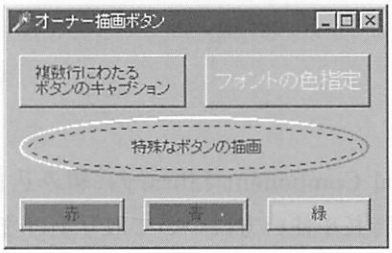
本章では、Visual Component Libraryに組み込まれているコンポーネントの特長や使い方について取り上げます。

Q. ボタンのキャプションに2行以上の文字列を表示させたいのですが、どうすればよいでしょうか。

Buttonは、キャプションに長い文字列が指定されても常に1行の文字列として認識します。

付録CDには、複数行でキャプションを表示したり、プログラムでボタンの表面を描画できるようにするButtonExコンポーネントが収録されています。ButtonExでは、Styleプロパティに bsOwnerDrawWithFrame を指定すると、キャプションに長い文字列や改行付きの文字列を指定したときに自動的に文字列を複数行で表示します。また、Buttonと違ってFontのColorプロパティ（文字色）の指定も反映されます。また、StyleがbsOwnerDrawかbsOwnerDrawWithFrameのときに、OnDrawButtonイベントハンドラを定義すれば、ボタンの表面をプログラムで描画できます。bsOwnerDrawの場合は、ボタンの輪郭もプログラムで描画します。

図4-1　オーナー描画ボタンの使用例



改行付きの文字列はオブジェクトインスペクタでは直接指定できません。プログラム実行時に改行付きの文字列を指定することもできますが、2Way-Toolの機能を使って設計時にも指定できます。このために、フォーム全体をスピードメニューの**[テキストとして表示(V)]**を選んでテキスト表示させ、目的のボタンの定義を探します。あるいは、目的のボタンを選んでから**[編集(E)｜切り取り(T)]**で切り取り、テキストエディタに**[編集(E)｜貼り付け(P)]**で貼り付けます。ボタンの定義は、次のようなものになります。

```
object ButtonEx1: TButtonEx
  Left = 100
  Top = 50
  Width = 75
  Height = 25
  Caption = 'ButtonEx1'
  TabOrder = 2
end;
```

ここで、Captionを定義している行を「Caption = 'Multi'#13#10'Line'」(#13#10は、復帰改行の文字)と修正し、ふたたび[フォームとして表示(V)]か、エディタ上での[編集(E)|切り取り(T)]とフォーム上での[編集(E)|貼り付け(P)]を使ってビジュアルデザイナに戻します。この操作で、改行指定付きの文字列を指定できます。

この方法はLabelなど、他のコンポーネントでも有効です。

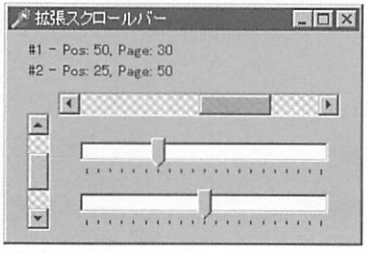
CHAP4¥USEBTNEX.DPR（ButtonExコンポーネントを使用）

Q. スクロールバーで、Delphiのコードエディタなどのようにつまみの幅を変更したいのですが、どうすればよいでしょうか。

ScrollBarコンポーネントには、つまみの幅を決めるプロパティはありません。

付録CDには、つまみの幅をPageプロパティで設定できるScrollBarExコンポーネントが収録されています。ScrollBarExでは、Pageが0以外の場合に、つまみの幅がスクロールバー全体のどれだけを占めるかを指定する意味になります。たとえば、Minが0、Maxが100のときにPageを50に設定すると、つまみの幅はスクロールバーの約半分になります。

図4-2　拡張スクロールバーの使用例



CHAP4¥USESBAR.DPR（ScrollBarExコンポーネントを使用）

Q. コンポーネントの大きさを少しずつ変えるようにプログラムしているのですが、途中の経過が表示されず最後の状態だけが表示されます。処理が速すぎるのでしょうか。

コンポーネントのHeightやWidthなどのプロパティを使って大きさを変更すると「再描画の要求」は発生しますが、実際に再描画されるわけではありません。たとえば、次のプログラムはフォーム上に配置したパネルの大きさを段々大きくしていますが、実際に表示されるのはこの処理が終わった後になるため、途中の経過は表示されません。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  for i := 10 to 99 do
    with Panel1 do
      SetBounds(Left, Top, i, i);
end;
```

次のようにUpdateメソッドを使えば途中の状態も描画されるようになります。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  for i := 10 to 99 do
    with Panel1 do
    begin
      SetBounds(Left, Top, i, i);
      Update;
    end;
end;
```

コンポーネントのUpdateの代わりにApplication.ProcessMessages;を呼び出しても、再描画されます。これは、Application.ProcessMessages;によってシステム中に残っているメッセージ（ここでは再描画メッセージ）を処理できるためです。

CHAP4¥ZOOMPNL.DPR

Q. 文字列グリッドで、選択中のセルの色を変更したいのですが、どうすればよいでしょうか。

文字列グリッドに、選択中のセルの色を変更するためのプロパティはありません。しかし、OnDrawCellというイベントハンドラを処理することにより、自分の好きな方法でセルの内容を描画できます。OnDrawCellは、セルの内容を描画するときに発生するイベントで、イベントハンドラは次のような形式をとります。

```
procedure DrawCell(Sender: TObject; Col, Row: Longint;
                   Rect: TRect; State: TGridDrawState);
```

ここで、Senderはイベントが発生したオブジェクト（StringGrid1など）、ColとRowは描画するセル、Rectは描画対象となる矩形領域、Stateはセルの状態をあらわしています。デフォルトと同じようにテキストを描画するためには次のようにします。

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject; Col, Row:Longint;
  Rect: TRect; State: TGridDrawState);
begin
  with StringGrid1.Canvas do
  begin
    Brush.Style := bsClear;
    TextRect(Rect, Rect.Left + 2, Rect.Top + 2,
             StringGrid1.Cells[Col, Row]);
  end;
end;
```

次のプログラム例は、選択範囲のセルについて背景色と文字色を黄色と赤に、入力フォーカスのあるセルの背景色と文字色を青と灰色にします。

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject; Col, Row:Longint;
  Rect: TRect; State: TGridDrawState);
begin
  with StringGrid1.Canvas do
  begin
    { 入力フォーカスを持っている場合 }
    if gdFocused in State then
    begin
      Brush.Color := clBlue;
      Font.Color := clSilver;
      TextRect(Rect, Rect.Left + 2, Rect.Top + 2,
               StringGrid1.Cells[Col, Row]);
    end
    { それ以外で、選択中のセルの場合 }
    else if gdSelected in State then
```

```
    begin
      Brush.Color := clYellow;
      Font.Color := clRed;
      TextRect(Rect, Rect.Left + 2, Rect.Top + 2,
               StringGrid1.Cells[Col, Row]);
    end;
  end;
end;
```

TGridDrawState型であらわされるState引数は、固定セルをあらわすgdFixed、選択中のセルであることをあらわすgdSelected、入力フォーカスのあるセルgdFocusedの状態が組み合わせで渡されます。これは、集合型として定義されていて、ある状態が有効かどうかは**in**という演算子を使って調べます。たとえば、固定セルかどうかはgdFixed **in** Stateで調べられます。

左側や上部にある固定セルは選択できないので、gdFixedが他の状態と組み合わせて指定されることはありませんが、gdSelectedとgdFocusedは組み合わせて指定されることがあります。固定セルでも選択範囲でもないセルは、どの状態にも該当しません。

このプログラムでgdFocusedよりも先にgdSelectedを判定してはいけません。範囲指定がある場合、入力フォーカスのあるセルは常にその範囲指定の中に入るため、gdSelectedを先に判定するとgdFocusedが判定されることがなくなってしまうためです。

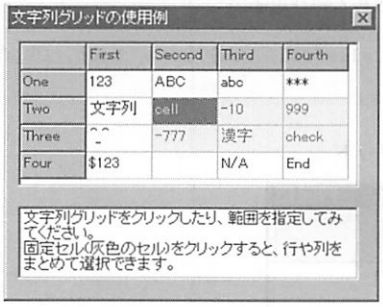
両方の**if**文の中で別々にテキスト出力(TextRect)を呼び出しているのは、どちらにも該当しない場合には何もしないようにするためです。本来、文字列グリッドはデフォルトでDefaultDrawingというプロパティがTrueになっており、すべてのセルに必要な情報を描画しています。OnDrawCellイベントで何かを描画することは、この上に描画することになり二度手間をかけることになります。再描画が不要な場所はできるだけ描画しないようにする方が実行速度が向上することになります。

DefaultDrawingプロパティをFalseにして、すべてのセルを自分で描画すれば二度手間による無駄をなくして好みのスタイルで描画することができます。ただし、DefaultDrawingをFalseにすると背景の塗りつぶし、色の設定、フォーカス（点線の枠）の表示など、すべてを自分で処理しなければならなくなります。DefaultDrawingプロパティがFalseのときに、デフォルトの動作と完全に置き換え可能なOnDrawCellイベントハンドラを以下に示します。このイベントハンドラで色の設定やフォントの設定を変更すれば、独自の表示方法を指定できます。

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject; Col, Row:Longint;
  Rect: TRect; State: TGridDrawState);
begin
  with StringGrid1.Canvas do
  begin
    Font := StringGrid1.Font;
    if gdFixed in State then
      Brush.Color := StringGrid1.FixedColor
    else if (gdFocused in State) or (State = []) then
```

```
      Brush.Color := StringGrid1.Color
    else
    begin
      Brush.Color := clHighlight;
      Font.Color := clHighlightText;
    end;
    TextRect(Rect, Rect.Left + 2, Rect.Top + 2,
             StringGrid1.Cells[Col, Row]);
    if (gdFixed in State) and StringGrid1.Ctl3D then
    begin
      Pen.Color := clBtnHighlight;
      Polyline([Point(Rect.Left, Rect.Bottom - 1), Rect.TopLeft,
                Point(Rect.Right, Rect.Top)]);
    end;
    if gdFocused in State then
      DrawFocusRect(Rect);
  end;
end;
```

図4-3　表示する色を変更した文字列グリッド



CHAP4¥STRGRID.DPR

Q. 文字列グリッドでセルごとに色を指定するには、どうすればよいでしょうか。

文字列グリッドでは、セルごとに色やフォントのスタイルを指定するプロパティはありません。したがって、セルの内容によって描画する色を変更するか、色のための2次元配列を定義して、その内容を使って描画することになります。

CHAP4¥STRGRID.DPR

Q. 文字列グリッドでセルの中に複数行に渡る文字列を表示させたいのですが、どうすればよいでしょうか。

デフォルトの描画ルーチンは単一行として表示するため、OnDrawCellで複数行を描画できるようにする必要があります。OnDrawCellを次のように記述すれば、文字列はセルの範囲に収まるよう複数行で表示されます。DefaultDrawingプロパティをFalseにする場合は、前述のイベントハンドラのように背景色の指定などをすべて処理する必要があります。

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject; Col, Row:
Longint;
  Rect: TRect; State: TGridDrawState);
begin
  { 背景を塗りつぶしておく }
  StringGrid1.Canvas.FillRect(Rect);
  { Windows API のテキスト描画関数を呼び出す }
  DrawText(StringGrid1.Canvas.Handle,
           PChar(StringGrid1.Cells[Col, Row]), -1, Rect, DT_WORDBREAK);
end;
```

図4-4　複数行を表示する文字列グリッド

CHAP4¥SGRIDML.DPR

Q. 文字列グリッドで固定セルをクリックして列や行全体を選択させたいのですが、固定セルをクリックしてもOnClickイベントが発生しないようです。

文字列グリッドの固定セルをクリックしてもOnClickイベントは発生しないので、OnMouseDownイベントを処理するようにします。次のプログラムは、FixedColsとFixedRowsプロパティがそれぞれ1のときに、固定セルをクリックして列や行全体を選択するイベントハンドラです。

```
procedure TForm1.StringGrid1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ACol, ARow: Longint;
  GRect: TGridRect;
begin
  with StringGrid1 do
  begin
    { クリックされた座標からセルの位置を取得する }
    MouseToCell(X, Y, ACol, ARow);
    GRect.Left := ACol;
    GRect.Right := ACol;
    GRect.Top := ARow;
    GRect.Bottom := ARow;
    { 左端の桁であれば、その右の範囲のすべてを選択領域にする }
    if (ACol = 0) then
    begin
      GRect.Left := ColCount - 1;
      GRect.Right := 1;
    end;
    { 先頭の行であれば、その下の範囲のすべてを選択領域にする }
    if (ARow = 0) then
    begin
      GRect.Top := RowCount - 1;
      GRect.Bottom := 1;
    end;
    { 左端の桁か先頭の行のときは、選択領域を反映させる }
    if (ACol = 0) or (ARow = 0) then
    begin
      Col := GRect.Right;     { フォーカス位置を混乱させないよう }
      Row := GRect.Bottom;    { あらかじめセルを移動しておく }
      Selection := GRect;
    end;
  end;
end;
```

CHAP4¥STRGRID.DPR

Q. マウスのクリック（OnClick）とダブルクリック（OnDblClick）で処理を変えたいのですが、ダブルクリックが発生する前に必ずOnClickが発生してしまいます。ダブルクリックしたときにクリックの処理をしないようにするには、どうすればよいでしょうか。

通常、ダブルクリックしようとしているかどうかに関わらず、最初のクリックで必ずOnClickイベントが発生します。これは、最初のクリックがダブルクリックの1回目かどうかを判断しようとすると、ダブルクリックを受け入れる時間の間、処理が先送りされてしまうためです。

付録CDには、クリックとダブルクリックを判別するための手続きMarkDoubleClickと関数CheckDoubleClickが提供されています。これらは、SysCtrlsユニットに含まれており、それぞれOnClickとOnDblClickイベントハンドラで使います。典型的なプログラムを以下に示します。

```
{ OnClick イベントハンドラ }
procedure TForm1.ScrollUpClick(Sender: TObject);
begin
  MarkDoubleClick(Sender);
end;

{ OnDblClick イベントハンドラ }
procedure TForm1.ScrollUpDblClick(Sender: TObject);
begin
  if CheckDoubleClick(Sender) then
  begin
    { ダブルクリックのための処理 }
  end
  else
  begin
    { クリックのための処理 }
  end;
end;
```

最初のクリックがダブルクリックの1回目かどうかを判断しなければならないため、クリックの処理はダブルクリックの待ち時間の分だけ先送りされます。ダブルクリックの待ち時間は、コントロールパネルのマウスで確認できます。プログラムで待ち時間を調べたり設定するためには、Windows APIのGetDoubleClickTime、SetDoubleClickTimeを使います。

CHAP4¥DBLCLK.DPR（SysCtrlsユニットを使用）

Q. Editコンポーネントをいくつか配置して、タブキーの代わりに矢印キーや〔Enter〕（リターンキー）で項目を移動させようと考えています。どのようにプログラムすればよいでしょうか。

タブキーの代わりに〔Enter〕を使うだけであれば、フォームのKeyPreviewプロパティをTrueにし、OnKeyPressイベントハンドラを次のように定義すればよいでしょう。

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then
  begin
    SelectNext(ActiveControl, True, True);
    Key := #0;
  end;
end;
```

ただし、フォーム上にデフォルトボタン（DefaultプロパティがTrueになっているボタン）がある場合は無効です。デフォルトボタンは、EditやListBoxなど〔Enter〕を受け取らないコントロール上で、〔Enter〕を押した場合に動作するボタンです。

特定のキーの動作を指定する場合、OnKeyDownまたはOnKeyPressイベントを処理します。どのキーに対してどちらのイベントを使う方が適当かという基準は、文字コードを発生するかどうかで判断します。たとえば、矢印キーはOnKeyDownイベントハンドラで処理できる仮想キーコードは発生しますが、OnKeyPressイベントは発生しません。〔Enter〕は、OnKeyPressイベントを発生し13という文字コードを持つ文字を渡します。

文字コードを発生するキー操作は通常の英数字、記号（、漢字）および〔Esc〕、〔BS〕、〔Enter〕です。これらは、OnKeyPressで処理します。ファンクションキー、〔Ins〕、〔Del〕、〔PageUp〕、〔PageDown〕、矢印キー、〔Home〕、〔End〕などの特殊キーは文字コードを発生しないのでOnKeyDownで処理します。

通常、〔Tab〕はコントロールを移動するキーとみなされ、イベントを発生しません。Editコンポーネントを図4-5のように配置した場合に、矢印キーや〔Enter〕でコントロールを移動できるようにしたプログラム例を以下に示します。

図4-5　複数のEditを配置したフォーム

このプログラムでは、すべてのEditコンポーネントのOnKeyDownとOnKeyPressイベントハンドラを共通に指定します。また、このためにレコード型TMoveInfoを定義して、それぞれのキー入力に対応する移動先情報をあらかじめ初期化しています。左右の矢印キーは、Editコンポーネント中のカーソルが左端や右端にあるときにだけ、別のコントロールに移動します。

```
type
  TMoveInfo = record
    { 自分自身、左、上、右、下、〔Enter〕に対応するコントロールの定義 }
    SelfCtrl, LeftCtrl, UpCtrl, RightCtrl, DownCtrl, EnterCtrl: TEdit;
  end;

var
  MoveInfo: array [1..4] of TMoveInfo;

procedure TForm1.FormCreate(Sender: TObject);
begin
  { フォームを作成するときに移動情報を初期化しておく }
  with MoveInfo[1] do
  begin
    SelfCtrl := Edit1;
    LeftCtrl := Edit3;
    UpCtrl := Edit2;
    RightCtrl := Edit3;
    DownCtrl := Edit2;
    EnterCtrl := Edit3;
  end;
  with MoveInfo[2] do
  begin
    SelfCtrl := Edit2;
    LeftCtrl := Edit4;
    UpCtrl := Edit1;
    RightCtrl := Edit4;
    DownCtrl := Edit1;
    EnterCtrl := Edit4;
  end;
  with MoveInfo[3] do
  begin
    SelfCtrl := Edit3;
    LeftCtrl := Edit1;
    UpCtrl := Edit4;
    RightCtrl := Edit1;
    DownCtrl := Edit4;
    EnterCtrl := Edit2;
  end;
  with MoveInfo[4] do
  begin
    SelfCtrl := Edit4;
    LeftCtrl := Edit2;
    UpCtrl := Edit3;
    RightCtrl := Edit2;
    DownCtrl := Edit3;
    EnterCtrl := Edit1;
  end;
end;

procedure TForm1.Edit1KeyDown(Sender: TObject; var Key: Word;
```

```
  Shift: TShiftState);
var
  i: Integer;
begin
  for i := Low(MoveInfo) to High(MoveInfo) do
    if Sender = MoveInfo[i].SelfCtrl then
    begin
      { 移動情報にしたがって、コントロールを移動させる }
      with MoveInfo[i] do
      begin
        if Key = VK_UP then
          UpCtrl.SetFocus
        else if Key = VK_DOWN then
          DownCtrl.SetFocus
        else if ((SelfCtrl.SelStart + SelfCtrl.SelLength) = 0)
                and (Key = VK_LEFT) then
          LeftCtrl.SetFocus
        else if (SelfCtrl.SelStart = Length(SelfCtrl.Text))
                and (Key = VK_RIGHT) then
          RightCtrl.SetFocus
      end;
      Break;
    end;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
var
  i: Integer;
begin
  if Key = #13 then
    for i := Low(MoveInfo) to High(MoveInfo) do
      if Sender = MoveInfo[i].SelfCtrl then
      begin
        MoveInfo[i].EnterCtrl.SetFocus;
        Key := #0;
        Break;
      end;
end;
```

CHAP4¥EDITMOV.DPR

Q. リストボックスで、選択中の文字列の色を変更したいのですが、どうすればよいでしょうか。

リストボックスの選択文字列は、Windowsのコントロールパネルで反転表示と反転表示の文字に設定されている色を使って表示されます。選択中のものなどリストボックスの項目を異なる色で表示したり表示形式を変更するためには、リストボックスのStyleプロパティをlbOwnerDrawFixedかlbOwnerDrawVariableに変更します。

lbOwnerDrawFixedを指定した場合、それぞれの項目の高さにはItemHeightプロパティに指定したものが使われます。項目はOnDrawItemイベントハンドラで描画します。項目ごとに高さが違う場合は、スタイルとしてlbOwnerDrawVariableを選択します。この場合は、OnDrawItemに加えてOnMeasureItemイベントハンドラを定義して、各項目に対応する高さを返します。いずれもイベントハンドラを定義していない場合は、デフォルトの描画ルーチンが使われます。

たとえば、選択中の文字の背景を黄色(clYellow)にしたい場合は、次のようにします。

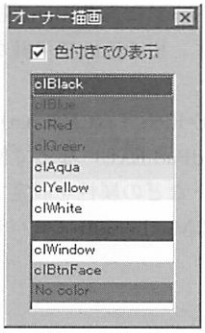
```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
  with ListBox1 do
  begin
    if odSelected in State then
      Canvas.Brush.Color := clYellow;
    Canvas.FillRect(Rect);
    Canvas.TextOut(Rect.Left + 2, Rect.Top, Items[Index]);
  end;
end;
```

次のプログラムは、リストボックスに入力されている色の名前（clBlueやclRedなど）に対応する色を背景として使用するプログラム例です。

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var
  Temp: TColor;
begin
  with ListBox1 do
  begin
    { ListBox1 の Font/Brush を使って Canvas の Font/Brush を設定 }
    Canvas.Font := Font;
    Canvas.Brush := Brush;
    try
      { 文字列名から色を取得し、背景色として指定する }
      Canvas.Brush.Color := StringToColor(Items[Index]);
    except
```

```
      { 文字列名が適切でない場合は、例外が発生する }
      on E:Exception do Canvas.Brush.Color := clRed;
    end;
    { Brush の色と文字列の色が同じ場合は、文字列の色を反転 }
    if Canvas.Brush.Color = Canvas.Font.Color then
      Canvas.Font.Color := $FFFFFF - ColorToRGB(Canvas.Font.Color);
    { 選択中の項目は、文字列の色と背景色を交換 }
    if odSelected in State then
    begin
      Temp := Canvas.Font.Color;
      Canvas.Font.Color := Canvas.Brush.Color;
      Canvas.Brush.Color := Temp;
    end;
    { 禁止状態の場合は、背景色を明るくする }
    if odDisabled in State then
      Canvas.Brush.Color := ColorToRGB(Canvas.Brush.Color) or $808080;
    { 背景の描画 }
    Canvas.FillRect(Rect);
    Canvas.Brush.Style := bsClear;
    { 文字列の描画 }
    Canvas.TextOut(Rect.Left + 2, Rect.Top, Items[Index]);
  end;
end;
```

図4-6　サンプルプログラムの実行画面



CHAP4¥ODLBOX.DPR

Q. コンボボックスで、ドロップダウンリストをプログラムで表示させることはできませんか。

A．ComboBoxコンポーネントには、DroppedDownというプロパティがあり、ドロップダウンリストが表示されているかどうかを確認したり、強制的に表示／消去することができます。たとえば、ドロップダウンリストを表示させるためにはComboBox1.DroppedDown := True;のようにします。

Q. Memoコンポーネントを使っていますが、32KB以上のファイルは編集できないのですか。

Windows95は32ビットOSですが、多くの部分で16ビットの機能がそのまま使われています。このため、MemoコンポーネントもWindows 3.1と同じく16ビットでの制約を受けています。

Delphi 2.0では、コンポーネントパレットのWin95ページにあるRichEditコンポーネントを使うことで32KBを越えるテキストを編集できます。RichEditで単純なテキストを編集したい場合は、PlainTextプロパティをTrueにします。

PlainTextがFalseの場合、リッチテキスト形式（RTF=Rich Text Format）という形式で編集することになります。これはワードパッド（WordPad.EXE）などで使われている形式です。この場合、文字単位・段落単位で文字フォントなどの属性を設定できます。

なお、RichEditで64KB以上のテキストを扱う場合はMaxLengthプロパティを編集したい最大値に設定しておく必要があります。

CHAP4¥BIGEDIT.DPR

Q. MemoやRichEditでテキストの最後にカーソルを移動させるには、どうすればよいでしょうか。

カーソル位置はSelStartプロパティで0から始まる文字単位で指定できます。また、MemoやRichEditのテキストの大きさはGetTextLenメソッドで取得できます。これらを使って次のようにすれば、カーソル位置をテキストの最後に移動できます。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.SelStart := Memo1.GetTextLen - 1;  { カーソル位置を末尾に移動 }
end;
```

CHAP4¥BIGEDIT.DPR

Q. MemoやRichEditでカーソルのある行番号を調べたり、指定した行番号にカーソルを移動させることはできませんか。

MemoやRichEditには、カーソル位置を行番号で取得したり設定するプロパティはありません。カーソル位置の行番号を取得するためには次のようにします（LineNoはInteger型の変数）。

```
with Memo1 do
  LineNo := SendMessage(Handle, EM_LINEFROMCHAR, SelStart, 0);
```

また、指定したカーソル位置に移動するには、次のようにします。

```
with Memo1 do
  SelStart := SendMessage(Handle, EM_LINEINDEX, LineNo, 0);
```

CHAP4¥BIGEDIT.DPR

Q. EditやMemoコンポーネントで挿入モードと上書きモードを切り換えることはできませんか。

EditやMemoコンポーネントが使っているWindowsのEDITコントロールには、上書きモードはありません。上書きモードを疑似的に表現するプログラム例を以下に示します。ここでは、フォームにMemoコンポーネントを配置して、OnKeyPress、OnKeyDownイベントハンドラを定義しています。

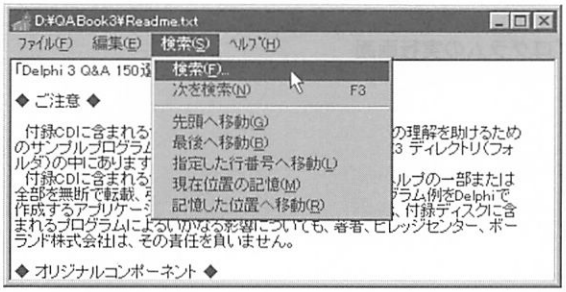
```
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    procedure Memo1KeyPress(Sender: TObject; var Key: Char);
    procedure Memo1KeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
  private
    FOverwriteMode: Boolean;  { 上書きモードのためのフラグ }
    ...
  end;
  ...
procedure TForm1.Memo1KeyPress(Sender: TObject; var Key: Char);
begin
  { 上書きモードで、表示文字が入力された場合は、}
  { Memo1 コンポーネントに対して Detele キーを送出する }
  if FOverwriteMode and (Ord(Key) >= $20) then
    with Memo1 do
    begin
      Perform(WM_KEYDOWN, VK_DELETE, 0);
      Perform(WM_KEYUP, VK_DELETE, 0);
    end;
end;

procedure TForm1.Memo1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  { Insert キーが押されたら、上書きモードを切り替える }
  if Key = VK_INSERT then
  begin
    FOverwriteMode := not FOverwriteMode;
    Key := 0;
  end;
end;
```

CHAP4¥BIGEDIT.DPR

Q. Memoコンポーネントで文字列の検索はどうすればよいでしょうか。

Memoコンポーネントには、文字列を検索するためのメソッドはありません。しかし、RichEditにはFindTextというメソッドがあります。RichEditのPlainTextプロパティをTrueにしてMemoの代わりに使うことで、通常のテキスト編集で検索機能を利用できます。FindTextは、大文字・小文字を同一視した検索やワード検索もできます。

図4-7　サンプルプログラムの実行画面



CHAP4¥BIGEDIT.DPR

Q. RichEditを使って書式指定付きの文章を編集させています。文字列の下付き指定は、フォントの大きさを小さくすればよいのですが、上付き指定はどうすればよいでしょうか。

通常、RichEditでの文章の編集では、文字列の下側（ベースライン）が常に同じ位置になります。上付き指定のためには、このベースラインを上にずらす必要がありますが、RichEditには、このための機能はありません。

付録CDには、ベースラインの指定ができるRichEditExコンポーネントが収録されています。RichEditExでは、SetBaseLineメソッドで選択範囲のベースラインをずらすことができます。たとえば、デフォルトのフォント（Fontプロパティ）の半分だけ上にずらす場合は、RichEditEx1.SetBaseLine(RichEditEx1.Size **div** 2);とします。

CHAP4¥RTEDIT.DPR（RichEditExコンポーネントを使用）

Q. Editコンポーネントで1行入力しているのですが、電卓のように右寄せで入力することはできないのですか？

Editコンポーネントが使っているWindowsのEDITコントロールは、1行入力の場合には右寄せができません。代わりにMemoコンポーネントで、1行目だけを使うことができます。このとき、WantReturnsとWantTabsプロパティはFalseにしておきます。

付録CDには、右寄せあるいは中央揃えに対応した1行入力ができるEditExコンポーネントが収録されています。EditExでは、Alignmentプロパティによってテキストを左右、中央のいずれかに合わせたり、3桁ごとにカンマ(,)を表示させることもできます。

図4-8　サンプルプログラムの実行画面

CHAP4¥CALCDEMO.DPR（EditExコンポーネントを使用）

Q. 入力ボックスなどで、かな漢字変換を使ったときに自動的にヨミガナを取り出すことはできませんか。

Delphiの標準的なコンポーネントには用意されていませんが、Windows APIを使って、かな漢字変換で使われた構成文字列（ヨミガナ）を取り出すことができます。

付録CDには、ヨミガナを取り出すためのCompoStringコンポーネントが収録されています。CompoStringは、EditやMemoなどの入力用コントロールと組み合わせて利用できます。まず、Controlプロパティに対象となるコンポーネントを指定します。次に、OnCompositionStrイベントに、ヨミガナが入力されたときのイベントハンドラを記述します。

たとえば、ControlにEdit1を指定し、OnCompositionStrに以下のイベントハンドラを定義すると、Edit1にかな漢字変換を使って入力するたびに、フォームのキャプションに

ヨミガナが追加されます。

```
procedure TForm1.CompoString1CompositionStr(Sender: TObject; Value:
string);
begin
  Caption := Caption + Value;
end;
```

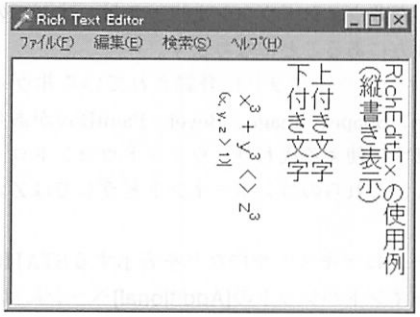
CHAP4¥USECOMPO.DPR（CompoStringコンポーネントを使用）

Q. 文章を縦書きで編集したいのですが、よい方法はないでしょうか。

日本語版Windows 95では、RICHEDITコントロールに縦書き編集の機能があります。

付録CDには、縦書き編集に対応したRichEditExコンポーネントが収録されています。RichEditExでは、Verticalプロパティを Trueにすることで縦書きで編集できるようになります。

図4-9　縦書き指定を使ったRichEditEx



なお、縦書き編集したテキストをファイル（.RTF）に保存する場合は、縦書きの情報も記録されます。このため、保存したファイルをワードパッドなどで読み込む場合でも縦書きとなります。

CHAP4¥RTEDIT.DPR（RichEditExコンポーネントを使用）

Q. **Memoコンポーネントの上にLabelコンポーネントを配置したいのですが、スピードメニューの[前面に移動]を選択してもLabelコンポーネントが上に表示されません。**

[前面に移動(F)]コマンドを使っても、Memoコンポーネントの上にLabelコンポーネントを表示することはできません。この仕組みを説明するために、コンポーネントのZオーダーについて解説します。

コンポーネントの前後関係は、Zオーダーという言葉で表現します。フォームの水平、垂直方向をX、Yにたとえると、Zは前後（深さ）をあらわし、Zオーダーが浅いほど、他のコンポーネントの上に重なって表示されます。**[前面に移動]**コマンドはコンポーネントのZオーダーを手前（前方）に移動し、**[背面に移動]**コマンドは最も奥（後方）に移動します。Zオーダーは、メニューやタイマーのような非ビジュアルコンポーネントでは関係ありません。

ウィンドウハンドルを持つコンポーネントをウィンドウコントロール、ウィンドウハンドルを持たないコンポーネントを非ウィンドウコントロールとします。非ウィンドウコントロールは、実際にはそのコントロールが配置されているコンポーネント（パネルやフォーム）上に描画されているに過ぎません。

このため、ウィンドウコントロールどうしでは前後関係を変更できても、非ウィンドウコントロールはウィンドウコントロールの関係を超えて、前後関係を変更することはできないのです。フォーム上に配置した非ウィンドウコントロールは、すべてのウィンドウコントロールよりもZオーダーが後方にあることになります。

Delphiのビジュアルコンポーネントライブラリに登録されている非ウィンドウコントロールには、Label、SpeedButton、Shape、Image、Bevel、PaintBoxがあります。これらは、ウィンドウハンドルを消費せずに使える代わり、ウィンドウコントロールの上に表示することはできません。もちろん、これらのコンポーネントどうしではZオーダーによって前後関係を指定できます。

Windowsの標準的なコントロールにテキストや枠などを表示するSTATICコントロールというものがあります。コンポーネントパレットの**[Additional]**ページには、STATICコントロールを使ったStaticTextというコンポーネントがあるため、これを使えば、他のウィンドウコントロール上に配置できます。ただし、StaticTextにはTransparent（透明）プロパティはありません。

CHAP4¥USETEXT.DPR

Q. 実行時にプログラムでコンポーネントのZオーダーを変更することはできますか。

ビジュアルコンポーネント（コントロール）には、BringToFrontとSendToBackというメソッドがあり、それぞれコンポーネントのZオーダーを前面／背面に移動できます。コンポーネントをZオーダーの途中に設定するメソッドはありませんが、複数のコンポーネントで順にこれらのメソッドを呼び出せば、最後に設定したものほどZオーダーが優先されることになります。

CHAP4¥ZORDER.DPR

Q. 実行時にコンポーネントのZオーダーを知ることはできますか。

フォームのControlsというプロパティには、フォーム上に直接配置されているすべてのビジュアルコンポーネント（コントロール）が含まれています。ここには、Zオーダーで後方のコンポーネントから順に格納されているため、もっとも最後のコントロールがZオーダーで最前面にあることになります。なお、パネル上に配置されているコントロールはフォームのControlsプロパティではなくパネルのControlsプロパティに格納されます。

なお、Controlsと違い、フォームのComponentsプロパティには、フォーム自身やパネル上に配置されているかどうかに関わらず、すべてのコンポーネント（非ビジュアルとビジュアルのすべて）が格納されています。

CHAP4¥ZORDER.DPR

Q. プログラムの実行中にコンポーネントを生成させたいのですが、どうすればよいでしょうか。

Visual Basicでは、実行時にコンポーネント（コントロール）を生成させるためにコントロール配列を使う必要がありますが、Delphiでは任意のコンポーネントをいつでも生成できます。コンポーネントの生成は、他のクラスオブジェクトを生成する場合と同じでCreateというコンストラクタを呼び出します。

以下に、フォーム上に2つのButtonコンポーネントを配置し、Button1を押すと新しいEditコンポーネントが生成され、Button2を押すと削除される例を示します。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  DynEdit: TEdit;
begin
  { すでにコンポーネントを生成している場合は、何もしない }
  if FindComponent('DynEdit') <> nil then
    Exit;

  DynEdit := TEdit.Create(Self);
  with DynEdit do
  begin
    Parent := Self;                { 親コンポーネントを指定する }
    SetBounds(100, 50, 89, 33);    { 位置と大きさを指定する }
    Caption := 'Dynamic';          { キャプションを指定する }
    Name := 'DynEdit';             { コンポーネント名を指定する }
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  DynEdit: TComponent;
begin
  { 動的に生成したコンポーネントを探す }
  DynEdit := FindComponent('DynEdit');
  { コンポーネントが見つかった場合（nil でなかった場合）、 }
  { コンポーネントを解放する }
  if DynEdit <> nil then
    DynEdit.Free;
end;
```

Button1Clickメソッドで使われている**with**文は、一つのオブジェクト（コンポーネント）に対する操作において、いちいちオブジェクト名を指定しなくてもよいようにするものです。たとえば、この部分は次のようにも記述できますが、設定する内容が増えれば増えるほど、いちいちDynEdit.と記述するのがわずらわしくなるでしょう。

```
DynEdit.Parent := Self;                {親コンポーネントを指定する}
DynEdit.SetBounds(100, 50, 89, 33); {位置と大きさを指定する}
DynEdit.Caption := 'Dynamic';          {キャプションを指定する}
DynEdit.Name := 'DynEdit';             {コンポーネント名を指定する}
```

コンポーネントを動的に生成する際には、注意すべき点がいくつかあります。

まず、必ずコンストラクタCreateを呼び出します。通常、コンストラクタの引数にはフォーム自身を指定します。フォームのイベントハンドラでは、Selfを使えばよいでしょう。コンストラクタの引数にForm1のように具体的な名前を指定することもできますが、同じフォーム型から複数のフォームの実体を作成する場合の動作に支障があります。この引数に指定したものは、コンポーネントのオーナー（所有者）となり、所有者が解放されるときに自動的にコンポーネントも解放されます。

フォームのイベントハンドラ以外でコンポーネントを生成したい場合には、コンポーネントがビジュアルでない場合に限りコンストラクタの引数にApplicationを指定することができます。Applicationオブジェクトは、アプリケーション全体を管理する影のコンポーネントです。Applicationオブジェクトそのものは目に見えるものではないため、ビジュアルコンポーネントの生成に使うことはできません。

EditやButton、Imageといったビジュアルコンポーネントを生成する場合は、Parentプロパティを設定しなければなりません。Parentプロパティは、生成するコンポーネントをどのコンポーネント（またはフォーム）の上に表示するかを決めるものです。Parent以外のほとんどのプロパティにはデフォルトの値が設定されますが、Parentプロパティが設定されない限りコンポーネントはどこにも表示されなくなります。

ビジュアルコンポーネントの場合は、位置や大きさを指定することも重要です。このとき、位置や大きさを指定するためにLeftやTopなどのプロパティを個別に指定することもできますが、SetBoundsを使う方がより便利です。これは、単に位置と大きさを一度に指定できるだけではありません。LeftやTopなどのプロパティは、変更するたびにコンポーネントの表示が変更されます。つまり、これらを個別に指定することはコンポーネントを初期化する際にチラつかせてしまうことになるのです。

もし、ボタンなどでデフォルトの大きさを使いたいという場合は、SetBounds(100, 50, Width, Height);のようにすればよいでしょう。**with**文を使っていれば、これはDynEdit.SetBounds(100, 500, DynEdit.Width, DynEdit.Height);のように解釈されます。

コンポーネント名をあらわすNameも重要です。設計時にフォームに配置したコンポーネントは、自動的にフォームのメンバ名と同じ名前が付けられますが、動的に生成したコンポーネントは明示的にNameプロパティを設定しなければ名前は空になってしまいます。名前が空の場合は、FindComponentなどでコンポーネントを見つけることができなくなります。

コンポーネント名は、キャプションなどと違ってObject Pascalの識別子として正しいものでなければなりません。つまり、先頭は英字かアンダーライン(_)ではじまり、その後ろは英数字かアンダーライン(_)が続くことになります。記号や漢字などは使えません。

イベントハンドラの設定は、他のプロパティの設定と同じでイベントハンドラ名を代入するだけです。DynEdit.OnClick := Button1Click;のように既存のイベントハンドラ名を使ってもかまいませんし、同じスタイルの（TObject型のSenderという引数を取る）メソッドを独自に定義して割り当てることもできます。

プログラムでコンポーネントを生成する場合は、必要なユニットを自分でUses節に追加する必要があります。たとえば、標準的なコンポーネントはStdCtrls、拡張されたものはExtCtrls、各種のボタン類はButtonsユニットが必要です。コンポーネントがどのユニットで定義されているかを確認するためには、オンラインヘルプを参照してください。

CHAP4¥DYNEDIT.DPR

Q. **フォームやPaintBoxコンポーネントのCanvasプロパティに描画するときは、直ちに描画した内容が反映されるのですが、Imageコンポーネントの Canvasプロパティを使って描画すると、描画し終わった後に内容が反映されるようです。これはなぜでしょうか。**

フォームやPaintBoxコンポーネントのCanvasプロパティは、スクリーンのデバイスコンテキストを対象にしています。したがって、描画メソッドを呼び出すことは、そのままスクリーンに描画することになります。このため、描画イベント（OnPaint）を定義しておかない限り、描画した内容が他のウィンドウなどで消されてしまうと、その内容は消えたままになります。

これに対して、ImageコンポーネントのCanvasはImageコンポーネントに割り当てられているビットマップをあらわすメモリデバイスコンテキストを対象にしています。Imageコンポーネントにアイコンやメタファイルが割り当てられているときは、Canvasは使えません。つまり、ImageコンポーネントのCanvasに描画することは、メモリ内のビットマップイメージを更新することになります。Imageコンポーネントに描画した後、イベントハンドラを終了して始めてImageコンポーネントが持つ内容がスクリーンに反映されます。

Imageコンポーネントへの描画はメモリデバイスコンテキストを対象にしているため、他のウィンドウでImageコンポーネントの内容が消されても、ふたたびImageコンポーネントが表示されれば、描画した内容が再度表示されます。

ただし、ごく限定的な目的以外では、この仕組みをウィンドウの再描画に利用することは、好ましくありません。ウィンドウの大きさと同じビットマップを表示するために大量のメモリを消費するためです。

CHAP4¥DRAWIMG.DPR

Q. **TimerコンポーネントのIntervalを1に設定して、1ミリ秒ごとに処理をさせたいのですが、もっと長い間隔でしかOnTimerが呼び出されないようです。**

TimerコンポーネントはWindows APIのSetTimerを使っていますが、最低でも約55ミリ秒間隔以下にすることはできません（DOS/Vの場合）。

付録CDには、マルチメディアタイマーを使った、より精度の高いMMTimerコンポーネントが収録されています。MMTimerコンポーネントは、Timerコンポーネントと同じように使えますが、1ミリ秒に近い間隔でOnTiemrイベントを発生します。また、Start、StopメソッドやElapsedTimeプロパティを使って、ストップウォッチの代わりに使うこともできます。

TimerおよびMMTimerに設定するIntervalは、最低限の時間間隔を示す目安であり、この間隔でイベントが発生することは保証されていません。

CHAP4¥STOPWATC.DPR

Q. メディアプレーヤーの内部エラーやデータベース編集中のエラーなど、フォームに配置したコンポーネントがプログラム部分以外で発生するエラーは、どのように処理すればよいでしょうか。

プログラム中で、発生する例外は**try**～**except**構文で明示的に処理できます。たとえば、フォーム上に1つのButtonコンポーネントと、3つのEditコンポーネントを配置して、Button1のOnClickイベントハンドラに除算を実行するプログラムを記述できます。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
begin
  try
    a := StrToInt(Edit1.Text);
    b := StrToInt(Edit2.Text);
    c := a div b;
    Edit3.Text := IntToStr(c);
  except
    on E:EConvertError do ShowMessage('Convert Error!');
    on E:EDivByZero do ShowMessage('Divide by zero!');
  end;
end;
```

このプログラムは、Edit1やEdit2に入力した内容が整数に変換できないものであれば、EConvertError（変換エラー）という例外を発生し、Edit2に0を入力した場合はEDivByZero（ゼロ除算）の例外を発生します。

README.TXTに記載されているとおり、98シリーズのWindows95では浮動小数演算例外を処理できないものがあります。

しかし、メディアプレーヤーでコンポーネントに対する動作だけで例外が発生したり、データベース項目で上限や下限を設定しているときに範囲外の数値を入力してしまった場合などは、こうしたプログラムでエラーを捕捉することができません。

アプリケーションの実行中に発生する例外で、プログラムで処理されないものに対しては、ApplicationオブジェクトのOnExceptionイベントが発生します。したがって、このイベントハンドラを定義すれば、プログラムで処理できない例外も捕捉できます。Applicationは目に見えないオブジェクトなので、イベントハンドラはすべてプログラムで代入する必要があります。OnExceptionイベントハンドラには、アプリケーションオブジェクトと例外オブジェクトが渡されます。

例外の判定は、例外処理の**except**部と同様、より詳細なものから優先します。たとえば、EDivByZeroはEIntErrorから派生しているのでEIntErrorを先に判定すると、EDivByZeroもこれに含まれることになります。

```
type
  TForm1 = class(TForm)
    ...
    procedure FormCreate(Sender: TObject);
  private
    procedure AppException(Sender: TObject; E: Exception);
  end;

  ...

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := AppException;
end;

procedure TForm1.AppException(Sender: TOBject; E: Exception);
begin
  if E is EDivByZero then              { ゼロ除算エラー }
    MessageDlg('Divide by zero', mtError, [mbOk], 0)
  else if E is EIntError then          { 一般整数エラー }
    MessageDlg('Other integer error', mtError, [mbOk], 0)
  else if E is EMCIDeviceError then  { MCI デバイスエラー }
    MessageDlg('MCI device error', mtError, [mbOk], 0)
  else if E is EConvertError then     { 変換エラー }
    MessageDlg('Convert error', mtError, [mbOk], 0)
  else                                { その他のエラー }
    MessageDlg('Other exception', mtError, [mbOk], 0);
end;
```

付録CDに収録されているPseudoAppコンポーネントを使うことで、OnExceptionイベントハンドラが定義しやすくなります。

CHAP4¥APPEXCEP.DPR

Q. PageControlで、実行時に特定のページを表示しないようにできますか。

PageControlのそれぞれのページは、TabSheetというコンポーネントで構成されています。TabSheetには、自分自身を表示するためのVisibleプロパティとタブ（見出し）を表示するためのTabVisibleプロパティがあります。これらをTrueやFalseにすることで、PageControlの各ページを表示するかどうか決められます。

CHAP4¥ACTPAGE.DPR

Q. TabControlやPageControlのタブを左右に割り当てることはできませんか？

通常、TabControl/PageControlでは、TabPositionプロパティの設定によってタブは上下にのみ表示します。

付録CDには、タブを左右に表示できるように拡張したTabControlEx、PageControlExコンポーネントが収録されています。これらは、TabPositionプロパティをtpeLeftやtpeRightにすることで、タブの表示位置を左右に指定できます。

CHAP4¥TABPAGE.DPR（TabControlEx、PageControlExコンポーネントを使用）

Q. TabControlやPageControlのタブで、複数行の文字列を表示することはできませんか。

通常、TabControl/PageControlでは見出しを単一行としてしか表示しません。

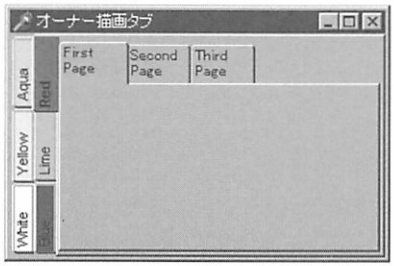
付録CDには、オーナー描画機能を追加したTabControlEx、PageControlExコンポーネントが収録されています。これらは、OwnerDrawプロパティをTrueにし、OnDrawTabイベントハンドラを定義することで、独自のプログラムコードを使ってタブを描画できます。

OnDrawTabは、リストボックスなどのOnDrawItemイベントに似ています。第1引数のControlは、必ずTTabControlEx、またはTPageControlExでキャストしてください。これらのクラスが、タブを描画するためのCanvasプロパティを持っています。

TabPositionプロパティをtpeLeftやtpeRightに設定して、タブを左右に割り当てている場合は、縦書きにしなければなりません。この場合、CreateFontIndirectなどを使って自分でフォントを作成しなければなりません。

OnDrawTabイベントハンドラで使われるTOwnerDrawState型は、StdCtrlsユニットで定義されているものです。このため標準コンポーネントを使っていない場合は、StdCtrlsユニットが取り込まれず、「未定義の識別子」というエラーが発生します。この場合は、ユニットソースコードの先頭の**uses**節にStdCtrlsを追加してください。

図4-10　オーナー描画を使ったTabControl/PageControl



CHAP4¥TABPAGE.DPR（Tab Control,Page Control Exコンポーネントを使用）

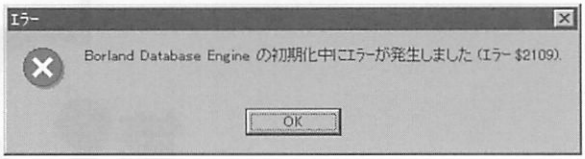
第5章

データベース

本章では、データベースコンポーネントの使い方やテーブルの操作についての質問を取り上げています。

Q. データベースアプリケーションを作成しようとしているのですが、次のようなエラーメッセージが発生してデータベースを利用できません。

図5-1　データベースエンジンのエラー



このエラーはデータベースエンジンが正しく初期化できなかったことをあらわしています。エラー番号の上位2桁はエラーの分類をあらわし、下位2桁がその詳細をあらわします。エラーの内容はDelphiのDOCディレクトリにあるBDE.INTの2802行目以降の内容で確認できます。

たとえば、$2109というエラー番号はエラーの分類としてはERRBASE_SYSTEM（システム関係の致命的なエラー）であり、その詳細を見るとERRCODE_CANTLOADIDAPI（IDAPIxx.DLLをロードできなかった）ということになります。

Borland Database Engine（BDE）のファイルは、インストール時に指定されたディレクトリ（C:¥Program Files¥Borland¥Common Files¥BDEなど）にコピーされます。Delphiのインストールに失敗していないかどうか、BDE環境設定が正常に動作するかどうか確認した上で、必要ならば再インストールしてください。

Q. DBGridで選択中のセルの色を変更したいのですが、どうすればよいでしょうか。

DBGridは文字列グリッド(StringGrid)と似ていますが、処理内容には大きな違いがあります。まず、DBGridではセルを描画する際にOnDrawCellではなくOnDrawColumnCellイベントが発生します。対応するイベントハンドラは、次のような形式になります。

```
procedure DrawColumnCell(Sender: TObject; const Rect: TRect;
    DataCol: Integer; Column: TColumn; State: TGridDrawState);
```

また、DBGridにはDefaultDrawDataCellというメソッドが用意されており、このイベントハンドラに渡される引数をそのまま使って、デフォルトと同じ描画処理ができます。あらかじめ、DBGridのDefaultDrawingプロパティをFalseにしておき、OnDrawColumnCellイベントに次のイベントハンドラを定義します（DefaultDrawingプロパティがTrueでもかまいませんが、同じ内容を二度描画することになります）。

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect:
TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
begin
  DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
end;
```

色を変更したい場合は、このメソッドを呼び出す前にCanvasプロパティのFontやBrushを変更しておきます。たとえば、フォーカスのあるセルを背景を水色、文字を太字の青で描画するためには次のようにします。

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect:
TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
begin
  if gdFocused in State then
  begin
    DBGrid1.Canvas.Brush.Color := clAqua;
    DBGrid1.Canvas.Font.Color := clBlue;
    DBGrid1.Canvas.Font.Style := [fsBold];
  end;
  DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
end;
```

図5-2　独自のスタイルでDBGridを描画

OptionsプロパティのdgRowSelectedがTrueの場合、イベントハンドラに渡されるState引数は、選択中の行の項目はすべてgdSelectedになり、さらに先頭の項目にはgdFocusedが渡されます。こうした状況を除けば、たいていはgdSelectedとgdFocusedは同じセルを指しています。

なお、タイトルバーや左側のインジケーターは「カラムセル」ではないので、このイベントハンドラでは処理できません。これらは、dgTitlesやdgIndicatorで表示しないようにできます。また、タイトルフォントの色やスタイルはTitleFontというプロパティで指定できます。

CHAP5¥DRAWCELL.DPR

Q. DBGridに異なるテーブルの項目を表示することはできますか。

比較的小規模のデータを表示するだけであればQueryコンポーネントを使って実現できます。これは、複数のテーブルに対するSQLの問い合わせを実行し、異なるテーブルの項目を一つのレコードとして利用するものです。データ件数が多くなると、処理時間がかかるようになります。

以下に、Delphiのサンプルデータを利用した実行例を示します。

1. フォームにQueryコンポーネントを配置し、DatabaseNameにDBDEMOSを指定します。
2. Query1のSQL文に以下の内容を入力します。これにより、ORDERS.DBのOrderNo、CustNo、およびCustNoに対応するCUSTOMER.DBのCompanyが得られます。

```
SELECT OrderNo, CustNo, "CUSTOMER.DB".Company
  FROM "ORDERS.DB", "CUSTOMER.DB"
  WHERE "ORDERS.DB".CustNo = "CUSTOMER.DB".CustNo
  ORDER BY OrderNo
```

3. Query1のActiveプロパティをTrueにします。これは、プログラムの実行中にQuery1.Open;とすることと同じです。
4. DataSourceとDBGridコンポーネントを配置し、DataSource1のDataSetプロパティにQuery1を、DBGrid1のDataSourceプロパティにDataSource1を指定します。

参照したい項目にインデックスが付いている場合は、Tableコンポーネントを使えます。Tableコンポーネントを使う場合は計算項目を定義し、OnGetTextイベントハンドラで対応する情報を文字列として取得します。Tableコンポーネントを使って、前述と同様の処理をするためには次のようにします。

1. フォームに2つのTableコンポーネントを配置し、両方のDatabaseNameプロパティにDBDEMOSを指定し、Table1とTable2のTableNameプロパティにORDERS.DBとCUSTOMER.DBを指定します。
2. Table1コンポーネントをダブルクリックして項目エディタを呼び出し、スピードメニューの**[追加の追加(A)]**でOrderNoとCustNo項目を追加します。さらに、**[項目の新規作成(N)]**でCompanyという名前の計算項目（String型、長さ=30）を追加します。
3. 項目エディタでCompanyを選び、オブジェクトインスペクタのEventsページでOnGetTextイベントをダブルクリックします。ここで、OnGetTextイベントハンドラを次のように定義します。

```
procedure TForm1.Table1CompanyGetText(Sender: TField;  var Text:
string;
  DisplayText: Boolean);
begin
  if Table2.FindKey([Table1CustNo.Value]) then
    Text := '[' + Table2.FieldByName('Company').AsString + ']';
end;
```

4. Table1とTable2のActiveプロパティをTrueにします。
5. DataSourceとDBGridコンポーネントを配置し、DataSource1のDataSetプロパティにTable1を、DBGrid1のDataSourceプロパティにDataSource1を指定します。

問い合わせを使うよりもやや複雑ですが、Tableコンポーネントを使う方が項目を編集できたり、処理が高速であるというメリットがあります。

文字列を設定するためにOnCalcFieldsイベントハンドラは使えません。

CHAP5¥DBGMULT.DPR

Q. DBGridで複数のレコードを選択することはできませんか。

DBGridのOptionsプロパティでdgMultiSelectをTrueにすることで、DBGridで複数レコードを選択できるようになります。複数のレコードを選択するには、〔Ctrl〕を押しながらマウスでクリックします。

選択したレコードは、SelectedRowsというプロパティで参照できます。複数のレコード選択を試すため、次の手順でフォームを作成します。

1. フォームにTable、DataSource、DBGridを配置します。
2. Table1のDatabaseNameをDBDEMOSに、TableNameをCOUNTRY.DB、Activeを Trueにします。
3. DataSource1のDataSetをTable1にします。
4. DBGrid1のDataSourceをDataSource1に、OptionsではdgMultiSelectをTrueにします。
5. ButtonとListBoxを配置して、Button1に以下のようなイベントハンドラを定義します。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 0 to DBGrid1.SelectedRows.Count - 1 do
  begin
    Table1.Bookmark := DBGrid1.SelectedRows[I];
    ListBox1.Items.Add(Table1.FieldByName('Name').AsString);
  end;
end;
```

プログラムを実行して複数のレコードを選択した後、ボタンを押すと選択したレコードの情報がリストボックスに表示されます。

CHAP5¥DBGMREC.DPR

Q. DBGridでスクロールバーを表示させないようにしたいのですが、どうすればよいですか。

DBGridでは、横幅に項目が収まり切らない場合に横スクロールバーが、高さにレコード数が収まり切らない場合に縦スクロールバーが表示されます。

付録CDには、表示するスクロールバーを選択できるDBGridExコンポーネントが収録されています。DBGridExコンポーネントのScrollBarsプロパティをssNoneにすれば、縦横のスクロールバーは表示されません。この他、ssVertical（縦スクロールバーのみ）、ssHorizontal（横スクロールバーのみ）、ssBoth（両方）の指定ができます。

図5-3　スクロールバーのないデータベースグリッド

CHAP5¥CHKDBG.DPR（DBGridExコンポーネントを使用）

Q. フォーム上にコンポーネントを配置せずにテーブルを利用したいのですが、どうすればよいのでしょうか。

Delphiのすべてのコンポーネントは、実行時に動的に生成できます。TableやQueryなどのコンポーネントも例外ではありません。

動的にコンポーネントを生成する方法については、第4章の項目を参照してください。

次のプログラムは、Delphiのサンプルデータ(DBDEMOS)にあるORDERS.DBで、すべてのレコードのAmountPaid項目を加算するプログラム例です。フォームにButtonとEditコンポーネントを配置しておいてください。このプログラムは、Button1のOnClickイベントハンドラとして記述します。また、Table、Queryを使うためにUses節にはDB、DBTablesユニットを追加しておきます。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TempTable: TTable;  { Table コンポーネントのための変数 }
  Sum: Double;        { 合計金額のための変数 }
begin
  TempTable := TTable.Create(Self);  { コンポーネントの動的生成 }
  with TempTable do                  { TempTable に対して作用させる }
  begin
    try
      DatabaseName := 'DBDEMOS';     { データベース名を設定する }
      TableName := 'ORDERS.DB';      { テーブル名を設定する }
      Open;                          { テーブルのオープン }
      First;                         { 先頭へ移動 }
      Sum := 0.0;                    { 合計金額の初期化 }
      while not EOF do               { 最後のレコードまで繰り返す }
      begin
                                     { AmountPaid 項目の値を加算する }
        Sum := Sum + FieldByName('AmountPaid').AsFloat;
        Next;                        { 次のレコードに移動する }
      end;
      Close;                         { テーブルを閉じる }
      Edit1.Text := Format('%m', [Sum]); { Edit ボックスに結果を表示 }
    finally
      Free;                          { コンポーネントの解放 }
    end;
  end;
end;
```

次のプログラムは、同じ処理をQueryコンポーネントを使ってSQL文で処理するものです。Buttonコンポーネントを配置し、Button2のOnClickイベントハンドラとして記述してください。

```
procedure TForm1.Button2Click(Sender: TObject);
var
  TempQuery: TQuery;  { Query コンポーネントのための変数 }
begin
  TempQuery := TQuery.Create(Self);  { コンポーネントの動的生成 }
  with TempQuery do                  { TempTable に対して作用させる }
  begin
    try
      DatabaseName := 'DBDEMOS';     { データベース名を設定する }
      SQL.Clear;                     { SQL 文をクリアする }
                                     { SQL 文を設定する }
      SQL.Add('SELECT SUM(AmountPaid) FROM "ORDERS.DB"');
      Open;                          { 問い合わせを実行する }
                                     { Edit ボックスに結果を表示 }
      Edit1.Text := Format('%m', [Fields[0].AsFloat]);
    finally
      Free;                          { コンポーネントの解放 }
    end;
  end;
end;
```

CHAP5¥DYNTABLE.DPR

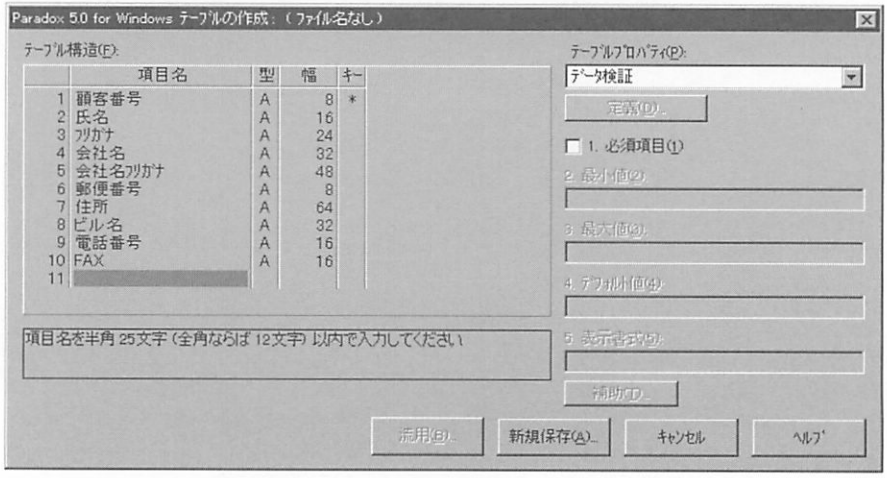
Q. 新しいテーブルを作成するには、どうすればよいでしょうか。

Delphi 3では、[ツール(T)｜データベースデスクトップ]でDatabase Desktopを呼び出せます。Database Desktopは、テーブルの作成、編集、QBE（例示による問い合わせ）などをサポートした簡単なデータベースアプリケーションであり、dBASE/Paradox形式をはじめ、各種のデータベーステーブルを作成できます。

テーブルを作成するためには、[新規作成(N)｜テーブル(T)]を選び（またはテーブルアイコンを右ボタンでクリックし[新規作成(N)]を選ぶ）、テーブルタイプを選択します。Paradox 7、Paradox 5.0 for Windows、Paradox 4、Visual dBASEなどといった項目がありますが、ここで選ぶ形式に関わらず新しい形式である必要がなければ旧式のものが選ばれることもあります。

テーブル形式を選ぶと、テーブルのために使いたい項目を入力するダイアログが表示されるので、必要な項目を入力します（図5-4）。右上のテーブルプロパティを使うと、必要に応じて二次インデックスやパスワードなどの設定もできます。最後に、[新規保存(A)]ボタンを押せば、新しい空のテーブルが作成されます。

図5-4　新しいテーブルのための項目を入力



	項目名	型	幅	キー
1	顧客番号	A	8	*
2	氏名	A	16	
3	フリガナ	A	24	
4	会社名	A	32	
5	会社名フリガナ	A	48	
6	郵便番号	A	8	
7	住所	A	64	
8	ビル名	A	32	
9	電話番号	A	16	
10	FAX	A	16	
11				

こうしたツールを使うのではなく、プログラムでテーブルを作成する場合は、Tableコンポーネントの CreateTable メソッドを使います。もっとも単純な方法は、次のようなものです。

1. フォームにTableコンポーネントを配置する。
2. DatabaseNameとTableNameプロパティを設定する。
3. 作成したいテーブルの種類をTableTypeプロパティで定義する。このとき、ttDefault以外を選択してテーブルの形式を明示する。
4. Table1コンポーネントをダブルクリックして項目エディタを呼び出す。
5. スピードメニューの**[項目の新規作成(N)]**ボタンで、適当な項目を定義する。
6. ボタンのOnClickイベントハンドラなどでTable1.CreateTable;とする。
7. プログラムを実行し、ボタンをクリックする。

[項目の新規作成(N)]ではさまざまなフィールド型を選べます。選んだフィールド型に対応して作成される項目型の対応表を以下に示します。

新規作成するときの型	Paradox	dBASE	ASCIIテキスト
String	文字型(A)	文字型(C)	Char
Integer	倍長整数型(I)	数値型(N)	Long Integer
Smallint	整数型(S)	数値型(N)	Number
Word	(使用不可)	(使用不可)	(使用不可)
Float	実数型(N)	数値型(N)	Float
Currency	金額型($)	数値型(N)	Float
BCD	BCD型(#)	数値型(N)	Float
Boolean	論理型(L)	論理型(L)	Bool
Date	日付型(D)	日付型(D)	Date
VarBytes	(使用不可)	(使用不可)	(使用不可)
Bytes	バイト型(Y)	(使用不可)	(使用不可)
Time	時間型(T)	文字型(C)	Time
DateTime	日付時間型(@)	文字型(C)	TimeStamp
Blob	バイナリ型(B)	メモ型(M)	(使用不可)
Memo	メモ型(M)	メモ型(M)	(使用不可)
Graphic	グラフィック型(G)	バイナリ型(B)	(使用不可)
AutoInc	カウンタ型(+)	(使用不可)	(使用不可)
作成できない型	書式付きメモ(F)	浮動型(F)	
	OLE型(O)	OLE型(O)	

TableコンポーネントでOLE型などがParadoxなどで作成されている場合、DelphiではBLObデータとして扱われます。これらは、Delphiのプログラムでは意味のある項目として利用できません。また、Delphi 3ではDBRichEditコンポーネントを使って書式付きメモ型（ftFmtMemo）を扱うことができますが、これはParadoxの書式付きメモ型と異なる形式（RTF）を使います。書式付きメモ型に限り、ParadoxとDelphiアプリケーションとの間の互換性はありません。

以下にテーブルの種類とDelphiのフィールド型の対応を示します。

Paradoxの項目型	Delphiのフィールド型
文字型(A)	TStringField
実数型(N)	TFloatField
金額型($)	TCurrencyField
整数型(S)	TSmallintField
倍長整数型(I)	TIntegerField
BCD型(#)	TBCDField
日付型(D)	TDateField
時間型(T)	TTimeField
日付時間型(@)	TDateTimeField
メモ型(M)	TMemoField
書式付きメモ型(F)	TBlobField
グラフィック型(G)	TGraphicField
OLE型(O)	TBlobField
論理型(L)	TBooleanField
カウンタ型(+)	TAutoIncField
バイナリ型(B)	TBlobField
バイト型(Y)	TBytesField

dBASEの項目型	Delphiのフィールド型
文字型(C)	TStringField
浮動型(F)	TFloatField
数値型(N)	TFloatField
日付型(D)	TDateField
論理型(L)	TBooleanField
メモ型(M)	TMemoField
OLE型(O)	TBlobField
バイナリ型(B)	TBlobField

ASCIIテーブルの型	Delphiのフィールド型
文字型(Char)	TStringField
浮動小数点型(Float)	TFloatField
16ビット整数型(Number)	TSmallintField
論理型(Bool)	TBooleanField
32ビット整数型(Longint)	TIntegerField
日付型(Date)	TDateField
時間型(Time)	TTimeField
日付時間型(TimeStamp)	TDateTimeField

CHAP5¥MKTABLE.DPR

Q. テーブルにインデックスを付けるにはどうすればよいでしょうか。

TableコンポーネントのAddIndexメソッドを使います。AddIndexは、次のような構文を持ちます。

```
procedure AddIndex(const Name, Fields: string; Options: TIndexOptions);
```

Nameはインデックス名、Fieldsは1つ以上の項目名または式(dBASEテーブルの場合)、Optionsはインデックスの属性をあらわします。ParadoxテーブルとdBASEテーブルのインデックスには、以下のような条件があります。ASCIIテーブルに対してはインデックスは付けられません。

Paradoxテーブルについて

・OptionsにixPrimaryを指定するとキーになります。キーがないテーブルでは、二次インデックスは指定できません。キーのインデックス名は空文字列（''）で、先頭の項目から連続したものだけを指定できます。
・複数の項目名をインデックスにする場合は、項目名をセミコロン（;）で区切ります。
・単一の項目名をインデックスにする場合は、インデックス名と項目名は同じものにします。
・つまり、項目名に式（ixExpression）は指定できません。
・インデックスは常にユニークになります。
・テーブルをオープンした時点では、キーがインデックスとして使われます。

dBASEテーブルについて

・キー（ixPrimary）はありません。Nameに空文字列は指定できません。
・大文字小文字を区別しない（ixCaseInsensitive）インデックスは指定できません。
・複数の項目名をインデックスにする場合は、式（ixExpression）を使います。このとき項目名は+で連結します。式インデックスを使う場合、FindKeyやFindNearestでは検索できません。SetKey、GotoKey、GotoNearestを組み合わせて検索してください。

次のプログラムは、Tableコンポーネントを動的に作成し、ID, Name、Addr、Telという項目を持つParadoxテーブルを作成し、さらにキーとしてIDを、二次インデックスとしてNameとAddrの組合せ、およびTelを作成するものです。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TempTable: TTable;  { Table コンポーネントのための変数 }
begin
  TempTable := TTable.Create(Self);  { コンポーネントの動的生成 }
  with TempTable do                  { TempTable に対して作用させる }
  begin
    try
      DatabaseName := 'DBDEMOS';     { データベース名を設定する }
      TableType := ttParadox;          { テーブルの種類を設定する }
      TableName := 'NEWTBLP.DB';     { テーブル名を設定する }
      with FieldDefs do                { 項目の設定 }
      begin
        Clear;
        Add('ID', ftInteger, 0, True);    { ID 項目(必須項目) }
        Add('Name', ftString, 20, True); { Name 項目(必須項目) }
        Add('Addr', ftString, 40, False);{ Addr 項目 }
        Add('Tel', ftString, 16, False); { Tel 項目 }
      end;
      CreateTable;                           { テーブルの作成 }
      AddIndex('', 'ID', [ixPrimary]);     { キー }
      AddIndex('NameAddr', 'Name;Addr', [ixCaseInsensitive]);
      AddIndex('Tel', 'Tel', []);
    finally
      Free;                              { コンポーネントの解放 }
    end;
  end;
end;
```

次のプログラムは、Tableコンポーネントを動的に作成し、Item、ID、Priceという項目を持つdBASEテーブルを作成し、インデックスとしてItem、ID、およびIDとItemの組合せを作成するものです。

```
procedure TForm1.Button2Click(Sender: TObject);
var
  TempTable: TTable;                   { Table のための変数 }
begin
  TempTable := TTable.Create(Self);  { コンポーネントの動的生成 }
  with TempTable do                    { TempTable に対して作用させる }
  begin
    try
      DatabaseName := 'DBDEMOS';      { データベース名を設定する }
      TableType := ttDBase;             { テーブルの種類を設定する }
      TableName := 'NEWTBLD.DBF';     { テーブル名を設定する }
      with FieldDefs do                 { 項目の設定 }
      begin
        Clear;
        Add('COMPANY', ftString, 20, False); { Company 項目 }
        Add('ITEM', ftString, 20, False);    { Item 項目 }
        Add('ID', ftInteger, 0, False);      { ID 項目 }
        Add('PRICE', ftCurrency, 0, False);  { Price 項目 }
      end;
      CreateTable;                        { テーブルの作成 }
      AddIndex('ITEMINDEX', 'ITEM', []);
      AddIndex('IDINDEX', 'ID', []);
      AddIndex('COMPITEM', 'COMPANY+ITEM', [ixExpression]);
```

```
    finally
      Free;                          { コンポーネントの解放 }
    end;
  end;
end;
```

CHAP5¥MKTABLE.DPR

Q. テーブルを異なる形式に変換するためには、どうすればよいでしょうか。

テーブルをコピーするためには、BatchMoveコンポーネントやTableコンポーネントのBatchMoveメソッドを使います。BatchMoveコンポーネントを使う場合、Sourceプロパティにコピー元になるTableかQueryコンポーネントを、Destinationプロパティにコピー先のTableコンポーネントを指定します。テーブルにレコードを追加する場合は、Modeプロパティをに、完全に置き換える（コピーする）場合はbatCopyを指定してください。

このとき、コピー先のTableのプロパティによって作成されるデータベースの型が決められます。TableTypeプロパティがttDefaultかttParadoxの場合は、Paradoxテーブルが作成され、ttDbaseのときはdBASEテーブルが、ttASCIIのときはASCIIテーブルが作成されます。

項目型にBLObを持つテーブルをASCIIテーブルにコピーしようとする場合など、対応できない型がある場合はテーブルはコピーできません。項目名や順序など項目の対応を変更したい場合は、BatchMoveコンポーネントのMappingsプロパティを設定します。BLOb項目のように変換できない項目はMappingsに記述しないようにします。なお、Tableコンポーネントに対して項目エディタで作成した項目オブジェクト（TxxxField）は参照されません。

CHAP5¥CONVTXT.DPR

Q. 固定長のテキスト形式のデータをdBASEやParadoxのテーブルに変換したいのですが、どうすればよいのでしょうか。また、カンマ区切りのデータを変換することはできますか。

固定長やカンマで区切られたテキスト形式のデータも、BDEによってテーブルとして利用できます。また、BatchMoveを使えば、dBASEやParadoxなど他のテーブル形式に変換することもできます。

テキスト形式のデータをASCIIテーブルとして使う場合の制約は、以下のとおりです。

・インデックスが使えない
・SQLの問い合わせが使えない
・レコードの削除やテーブルの途中への挿入ができない
・参照の整合性が使えない
・BLOb(Binary Large Object)が使えない
・区切り文字を使っているASCIIテーブルでは、レコードを編集できない

ASCIIテーブルを扱うには、スキーマファイル(.SCH)が必要です。スキーマファイルは、以下のようなものです。

```
[EXASCII]                    // ファイル名
Filetype=Delimited           // ファイル形式: Delimited または Fixed
Charset=ascii                // 言語ドライバ名(ascii)
Delimiter="                  // 文字列を囲む文字
Separator=,                  // 区切り文字
Field1=Name,Char,12,0,0      // 最初の項目
  ...
```

FiletypeにはFixedかDelimitedを指定し、それぞれフィールドを固定長にするかカンマなどの文字で区切ることをあらわします。CharSetには言語ドライバ（通常はASCII）を指定します。DelimiterとSeparatorは、FiletypeにDelimitedを指定した場合に使われる文字列を囲む文字とフィールドを区切る文字を指定します。

Field#（#は1,2,3...）は個々のフィールドの定義をあらわし、フィールドの名前、型名、幅、精度（FLOATのみ）、オフセット（Fixedテーブルのみ）の順で定義するものです。型名にはCHAR、FLOAT、NUMBER、BOOL、LONGINT、DATE、TIME、TIMESTAMPが使えます。

前述の方法でテーブルの作成した場合や、TableコンポーネントのBatchMoveメソッド、BatchMoveコンポーネントのExecuteメソッドなどを使ってASCIIテーブルを作成した場合には、固定長のASCIIテーブルに対するスキーマファイルが自動的に作成されます。

他のアプリケーションなどで作成したASCIIテキストを読み込むためには、対応するスキーマファイルを作成しておく必要があります。たとえば、20桁の文字列、16桁の文字

列、1桁の論理型、3桁の16ビット整数に対する固定長テキストデータに対するスキーマファイルは次のようになります。

```
[PERSON]
Filetype=Fixed
Field1=Name,Char,20,0,0
Field2=Tel,Char,16,0,20
Field3=Sex,Bool,1,0,36
Field4=Age,Number,3,0,37
```

CHAP5¥CSVTODB.DPR

Q. **Paradoxテーブルで複数の項目をインデックスとして定義しています。SetKeyとGotoKeyを使ってレコードを検索しようとしているのですが、最初の項目だけを指定しても2番目以降の項目を無視できません。**

TableコンポーネントのSetKeyメソッドを呼び出すと、いったんすべての項目が空に設定され、指定されているインデックスに含まれるすべての項目が検索対象となり、明示的に値が指定されない項目は空の値を検索しようとします。もし、最初の項目だけを検索対象にしたい場合はKeyFieldCountプロパティに1を代入しておきます。

たとえば、次のようなテーブルがありTable1コンポーネントに割り当てられており、PRODUCT、NUMBERにキーが指定されているとします。

レコード番号	PRODUCT	NUMBER
1	BC++	1
2	BC++	2
3	Delphi	
4	Delphi	1
5	Delphi	2
6	TC++	2
7	TC++	3

'TC++' という文字列で6番目のレコードを検索させるためには、次のようにプログラムします。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Table1 do
  begin
    SetKey;
    FieldByName('PRODUCT').AsString := 'TC++';
    KeyFieldCount := 1;
    GotoKey;
  end;
end;
```

SetKey、項目の設定、GotoKeyという一連の処理は、FindKeyメソッドを使えばより簡単に実現できます。次のプログラムも同じ処理をします。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.FindKey(['TC++']);
end;
```

FindKeyは項目名を使わず、インデックスの最初の項目から順に値を指定します。検索の対象となるのは指定された項目だけで、指定されていない項目は無視されます。

Q. SetRangeStart、SetRangeEnd、ApplyRangeを使ってテーブルの表示範囲を指定しているのですが、複数項目をインデックスにしている場合、範囲指定に使っていない項目が無視されません。

テーブルの範囲の設定についても、SetKeyと同様に項目の数を指定していない場合は、指定されているインデックスのすべての項目が範囲の対象となります。たとえば、前ページのテーブルで次のプログラムを実行すると、3番目のレコードのみが表示されます。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Table1 do
  begin
    SetRangeStart;
    FieldByName('PRODUCT').AsString := 'Delphi';
    SetRangeEnd;
    FieldByName('PRODUCT').AsString := 'Delphi';
    ApplyRange;
  end;
end;
```

もし、ApplyRangeの直前にKeyFieldCount := 1;という文を追加すれば、先頭の項目だけが範囲指定の対象と見なされ、3、4、5番目のレコードが表示されます。

SetRangeStart、項目の設定、SetRangeEnd、項目の設定、ApplyRangeの一連の処理は、SetRangeメソッドを使えばより簡単に実現できます。次のプログラムも同じ処理をします。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.SetRange(['Delphi'], ['Delphi']);
end;
```

図5-5　範囲指定を使ったサンプル

CHAP5¥TBLRANGE.DPR

Q. BDE環境設定ユーティリティ以外で、アプリケーション専用のエリアスを使いたいのですが、どうすればよいでしょうか。

Databaseコンポーネントを使えば、アプリケーション専用のエリアスを定義できます。Databaseコンポーネントを配置し、DatabaseNameプロパティに新しいエリアス名を、DriverNameをSTANDARDに指定します。さらに、Paramsプロパティに「PATH=パス名」と指定しておけば、他のTableやQueryコンポーネントは、このエリアス名を使ってテーブルや問い合わせを参照できます。

CHAP5¥DBALIAS.DPR

Q. テーブルから指定した項目に一致するレコードを取り出すために、Queryコンポーネントで次のようなSQL文を設定しています。

```
SELECT * FROM "ITEMS.DB" WHERE OrderNo = "1111"
```

しかし、テーブルが大きくなるほど処理が遅くなります。高速化することはできないでしょうか。

SQL文ではWHERE句に条件式を指定したり、_（任意の位置文字）や%（任意の文字列）を使うこともできるため、汎用的な条件付きの検索には便利です。しかし、基本的にテーブルのすべてのレコードを検査して問い合わせを実行するため、テーブルの大きさに比例して処理時間がかかります。

単純な文字列の一致や先頭が一致する項目の検索は、検索項目にインデックスを付けておきTableコンポーネントのFindKeyやFindNearestを使う方が高速に処理できます。たとえば、OrderNo項目で検索し、ItemNo項目の順に並べたい場合は、OrderNo、ItemNoに連結インデックス（ここではキー）を設定しておき次のようにプログラムします。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { 見つかったレコードの ItemNo をリストボックスに追加する }
  ListBox1.Items.Clear;
  with Table1 do
  begin
    { 最初の項目のみで検索する }
    if FindKey(['1111']) then
      { 一致するレコードがあったら、項目の値が一致している間 }
      { 次のレコードを調べる }
      while FieldByName('OrderNo').AsString = '1111' do
      begin
        { 必要な項目の情報を利用する }
```

```
        ListBox1.Items.Add(FieldByName('ItemNo').AsString);
        Next;
      end;
  end;
end;
```

項目の部分一致を利用したい場合でも、先頭の文字列が決まっている場合はFindNearestを使って同様の処理が実現できます。

CHAP5¥FINDREC.DPR

Q. **DataSourceコンポーネントにTableやQueryを割り当てて使っているのですが、対象となるテーブルや問い合わせのレコードを移動させるためのメソッドはDataSourceにはないのですか。**

DataSourceのDataSetプロパティを使います。DataSetは、Table、Query、StoredProcを設定できる汎用的なプロパティです。このDataSetプロパティは、単に対象となるデータセットを指定するだけでなく、これらのコンポーネントに共通するメソッドやプロパティを持っています。

Delphiではプロパティがコンポーネントをあらわしている場合、そこからメソッドやプロパティが扱えるようになっています。フォームやPaintBoxのCanvasプロパティに描画用のさまざまなメソッドが用意されているのと同じです。

たとえば、DataSourceを使ってデータ処理をしている場合、DataSource1.DataSet.Prior;やDataSource1.DataSet.Next;とすれば、DataSource1に指定されているテーブルや問い合わせのレコードを前後に移動できます。また、DataSource1.DataSet.Activeを調べることで、テーブルや問い合わせがアクティブであるかどうかを確認できます。

CHAP5¥NAVDSET.DPR

Q. レコードを前後に移動するためDBNavigatorのようにグループ化されたものではなく独立したボタンを作りたいのですが、どうすればよいでしょうか。

付録CDには、DBNavigatorの10種類のレコード操作（先頭／前／次／末尾レコードへの移動、挿入、削除、編集、登録、キャンセル、更新）に対応する個別のボタンを作成するためのDBNavButtonが収録されています。このコンポーネントは配置するたびに、新しい機能を持つボタンとして設定されます。ただし、対象となるデータソースを考慮しませんので、複数のデータソースに対して使う場合は配置した後でNavTypeプロパティを設定してください。

図5-6　DBNavButtonの使用例

CHAP5¥USENAVB.DPR（DBNavButtonコンポーネントを使用）

Q. dBASE形式のテーブルを使っていますが、レコードを削除してもテーブルの大きさが小さくなりません。

dBASE形式のテーブルは、レコードを削除しても内部的に削除フラグが追加されるだけで、実際のファイルからデータは取り除かれません。dBASE形式のテーブルから削除レコードを取り除くためには、Borland Database Engineの機能を使ってテーブルを圧縮しなければなりません。また、Paradoxテーブルでもメモ項目がある場合などは、レコードを削除しても再構築するまでファイルサイズが小さくなりません。

付録CDには、dBASE/Paradox形式のテーブルを圧縮できるTableExコンポーネントが収録されています。通常どおり、TableExにテーブルを割り当て、PackTableメソッドを呼び出します。

PackTableでは、dBASE/Paradoxテーブル以外では無効です。また、圧縮したいテーブ

ルが他のアプリケーションでオープンされている場合には、エラーが発生します。特に、Delphiの設計画面でTableExのActiveプロパティをTrueにしている場合にも同じ扱いになりますので、注意してください。

CHAP5¥PACKDB.DPR（TableExコンポーネントを使用）

Q. 暗号化されたdBASEテーブルを使いたいのですが、パスワードはどのように指定すればよいでしょうか。

Tableコンポーネントでは、暗号化されたdBASEテーブルを扱うことはできません。付録CDには、暗号化テーブルを扱えるようにするTableExコンポーネントが収録されています。TableExのAllowDBaseLoginプロパティをTrueにすると、暗号化されたdBASEを開こうとする際にOnDBaseLoginイベントが発生します。このイベントハンドラで、正しいグループ名、ユーザー名、パスワードを設定すれば、暗号化されたdBASEテーブルをオープンできます。

たとえば、次のように記述できます。

```
uses DBLogDlg;

procedure TForm1.TableEx1DBaseLogin(Sender: TObject; var UserName,
    GroupName, UserPassword: string);
begin
  GroupName := 'TESTGROUP';
  LoginDialog(TableEx1.TableName, UserName, UserPassword);
end;
```

CHAP5¥PACKDB.DPR（TableExコンポーネントを使用）

Q. ネットワークを使って複数のユーザーが同じテーブルを使っています。Paradoxでは、あるユーザーがデータを更新すると別のユーザーの画面も更新されたのですが、DelphiのDBGridなどでは更新されません。どうすればよいでしょうか。

Tableコンポーネントには、リモートでのデータ更新を検出する機能はありません。データを読み込み直せば更新されるため、適当なタイミングでTableのRefreshメソッドを呼び出すことで対応できます。

しかし、Borland Database EngineにはParadoxテーブルに限り、リモート更新を検出する機能があります。付録CDには、この機能を使ったTableExコンポーネントが収録されています。TableExはTimerコンポーネントとともに使い、TableExのTimerコンポーネントにTimerコンポーネントを指定すると、Timerに指定されている間隔でテーブルがリモートで更新されているかどうかを調べ、必要に応じてRefreshを呼び出します。

TableExコンポーネントは、リモートでの更新のみを検出します。スタンドアロン環境での更新には対応していません。

CHAP5¥CHKDBEX.DPR（TableEx、QueryExコンポーネントを使用）

Q. Queryを使った問い合わせ中に現在の進行状況を表示させたいのですが、どうすればよいでしょうか。

QueryのOpenメソッドは、問い合わせが終了するまで制御を返しません。付録CDには、Borland Database Engineの機能を使ってローカルテーブルの問い合わせ処理中に、イベントを発生させるQueryExコンポーネントが収録されています。

QueryExは、ローカルテーブルの問い合わせ中にOnProgressイベントを発生します。OnProgressイベントには、進行状況をあらわすメッセージ「レコードが追加されました:<数値>」と処理を継続するかどうかを決めるAction引数が渡されます。通常、<数値>には問い合わせの結果得られるレコード数をあらわします。進行状況が全体の何%に相当するかを調べることはできませんので、必要ならば対象となるテーブルのレコード数などを別途調べなければなりません。

このイベントはデータベース処理中に呼び出されます。このため、イベントハンドラからさらにデータベースを扱うことはできません。また、LabelやProgressBarなどのプロパティを設定する場合でも、Updateメソッドを呼び出さなければ表示は更新されません。さらに、問い合わせ処理中はメッセージを受け取らないため、キャンセルボタンなどを定義したい場合は、ボタンなどのメッセージを処理するためApplication.ProcessMessages;

を呼び出す必要があります。

前ページのTableExを使っている場合などは、Application.ProcessMessages;の呼び出しにより、暗黙のうちにタイマーイベントが発生しTableExの内部処理が行なわれます。これは、OnProgessイベントでデータ操作をしているのと同じです。TableExには、一時的にタイマーを抑止してApplication.ProcessMessages;を呼び出す、ProcessMessagesメソッドが定義されていますので、必要に応じてこれを使ってください。

図5-7　問い合わせ中の進行状況の表示

CHAP5¥CHKDBEX.DPR（TableEx、QueryExコンポーネントを使用）

第6章

for Visual Basic プログラマ

本章では、Visual Basicの開発者がDelphiを使う際の質問について取り上げています。また、Visual BasicとDelphiのプログラムの併用についても紹介しています。

※本章では、Visual BasicとはVisual Basic 4.0（32ビット版）または5.0を指します。

Q. Visual Basicの演算子に対応するObject Pascalの演算子には、どのようなものがありますか。

Object Pascalには、Visual Basicのすべての演算子に対応するものがあるわけではありません逆に、Object Pascalだけで使える演算子もあります。

たとえば、Visual Basicではべき乗(^)は演算子ですが、Object Pascalでは関数(Exp)を使う必要があります。このような注意は必要ですが、演算子や多くの組み込み手続きがVisual Basicの演算子の代わりに利用できます。

以下に、Visual Basicの演算子に対応するObject Pascalの演算子や関数を示します。これらは、機能が完全には一致しないことがあります。また、演算子の優先順位は、必ずしもVisual Basicと同等ではありません。

演算子の優先順位は、オンラインヘルプの「演算子/優先順位/式」を参照してください。

演算子	Visual Basic	Object Pascal
指数	^	Exp(y * Ln(x))
単項マイナス	-	-
乗算	*	*
除算	/	/
整数除算	¥	div
剰余	%	mod
加算	+	+
減算	-	-
文字列連結	+	+
比較	<	<
	<=	<=
	>	>
	>=	>=
	=	=
	<>	<>
文字列比較	Like	(対応するものなし)
オブジェクト比較	Is	=
論理NOT	Not	not
論理AND	And	and
論理OR	Or	or
論理XOR	Xor	xor
論理等価	Eqv	(=)
論理包含	Imp	(対応するものなし)

Q. Visual BasicのDoEventsの代わりに何を使えばよいのでしょうか。

Visual BasicのDoEventsは、時間のかかる繰返し処理などで他のメッセージが受け付けられなくなるのを避けるために使うものです。Delphiでは、同様の処理をするためにApplicationオブジェクトのProcessMessagesというメソッドを使います。

たとえば、処理の進行状況を示すメッセージダイアログのようなものを考えるために次のようなアプリケーションを作成します。

1. 空のプロジェクトを新規作成し、Form1にButtonコンポーネントを配置します。
2. 新しい空のフォームを新規に作成し（Form2）、Labelコンポーネントを配置します。
3. Form1に戻って、Button1のOnClickイベントに次のプログラムを割り当てます。

```
uses
  Windows, Messages, SysUtils, ..., StdCtrls, Unit2;
  ...

{ 期待通りに動作しないプログラム }
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  Form2.Show;                { Form2 を表示する }
  for i := 0 to 999 do
  begin
    { Form2 の Label に i を文字列として設定する }
    Form2.Label1.Caption := IntToStr(i);
  end;
  Form2.Hide;                { Form2 を非表示にする }
end;
```

このプログラムは、フォームのボタンを押すとForm2が表示されて、Label1コンポーネントに0～999までの数字を順に表示することを期待していますが、実際にはForm2には何も表示されません。

Form2.Show;としたときにForm2自身はすぐ描画されるのですが、Form2に配置されたコンポーネントは描画を要求するウィンドウメッセージが送られるだけです。しかし、この繰返し文の中ではウィンドウメッセージを処理する場所がないので、Form2には何も表示されないのです。

ここで、Form2.Label1.Caption := IntToStr(i);の下にApplication.ProcessMessages;という文を挿入します。ProcessMessages;とはアプリケーションに渡されているメッセージがあれば順に処理するというものです。つまり、描画要求があれば、必要なコンポーネントに正しくメッセージが伝達されます。これによって、進行状況が表示されるようになります。

CHAP6¥PROCMSG.DPR

Q. Visual Basicのコントロール配列に相当する機能は、どのように実現すればよいのでしょうか。

Delphiには、Visual Basicのコントロール配列にそのまま対応する機能はありません。コントロール配列は便利な点もありますが、オブジェクト指向プログラミングにおいてはオブジェクトの独立性に反するという面もあります。また、基本的にコントロール配列がなければできないという処理はありません。実際、Visual Basicでコントロール配列を利用する目的には「イベントハンドラの共有」「実行時のコントロールの生成」「配列全体に対する処理」などがありますが、これらのすべてについてDelphiではより柔軟にプログラミングできます。

まず、イベントハンドラの共有についてDelphiでは異なるコンポーネントでイベントハンドラを簡単に共有できます。Delphiでは、フォーム上で〔Shift〕を押しながらコンポーネントをクリックすることで複数のコンポーネントをまとめて選択できます。この状態でオブジェクトインスペクタのEventsページで適当なイベントをダブルクリックするか、定義済みのイベントを選ぶことで、選択したすべてのコンポーネントに対して同じイベントハンドラを使えます。

Visual Basicでは、コントロール配列に対するイベントハンドラの引数としてコントロールのインデックスが渡されますが、Delphiではコンポーネントそのものが引数となります。

たとえば、Visual Basicで、Btnというボタンのコントロール配列があるとき、フォームのタイトルバーにクリックしたボタンのキャプション文字列を割り当てるには次のようにします。

```
Sub Btn_Click(Index as Integer)
  Caption = Btn(Index).Caption
End Sub
```

ここで、Btn(Index)によってクリックしたボタンをあらわしています。Delphiで、これと同じ処理をするためには次のようになります。

```
procedure TForm1.BtnClick(Sender: TObject);
begin
  Caption := TButton(Sender).Caption;
end;
```

BtnClickに渡されるのはインデックスではなくイベントを発生したコンポーネントオブジェクトそのものです。引数を受け取るために汎用性のある型TObjectが使われていますが、TObjectにはCaptionというプロパティがないので、TButton(Sender)と型変換しています。Btn(Index)ではBtnがコントロール配列の名前であり、Btn(Index)で個々のコント

ロール要素を指し示しているという点の違いについて混同しないようにしてください。

BtnClickに渡される引数に汎用性を持たせているのは理由があります。Delphiでは、イベントハンドラを共有できるのは同じ種類のコンポーネントだけではないのです。たとえば、通常のボタン(Button)やビットマップ付きボタン(BitBtn)のOnClickイベントハンドラは、スピードバーのために使うスピードボタン(SpeedButton)、メニュー項目(MenuItem)のOnClickイベントハンドラとも共有できます。つまり、これらのOnClickイベントはすべてTObject型のSenderという引数を受け取るのです。

ただし、こうした異なる種類のコンポーネントでイベントハンドラを共有している場合は、単純にTButton(Sender).CaptionとしてCaptionプロパティを取り出すことはできません。Delphiには、実行時型情報という機能でオブジェクトの型を調べられます。あるオブジェクトがあるクラス（またはその下位クラス）であるかどうかを判断するには、**is**という予約語が使えます。たとえば、前述のイベントハンドラは次のように書き直せます。

```
procedure TForm1.BtnClick(Sender: TObject);
begin
  { イベントを発生したもの(Sender)の種類を調べる }
  if Sender is TButton then
    Caption := 'Button - ' + TButton(Sender).Caption
  else if Sender is TSpeedButton then
    Caption := 'SpeedButton - ' + TSpeedButton(Sender).Caption
  else if Sender is TMenuItem then
    Caption := 'MenuItem - ' + TMenuItem(Sender).Caption
  else
    Caption := 'Form1';
end;
```

このイベントハンドラでは、イベントを発生したコンポーネントオブジェクトSenderがButtonかSpeedButtonかMenuItemのうちの、どのクラスのものかを判断し、コンポーネントの種類とCaptionプロパティをフォームのタイトルバーに割り当てています。ただし、通常はスピードボタンにはキャプションを指定しません。また、クラス名には先頭にTがつく点に注意してください。

しかし、イベントハンドラが呼び出されたときに、Visual Basicのようにコンポーネントを数値で区別したいこともあるでしょう。このためにはTagプロパティを使うことができます。Tagプロパティは、すべてのコンポーネントのプロパティとして定義されているもので、Longint型で定義されています。これは、Delphi自身では使われないため、プログラマーが自由に意味を持たせることができます。

たとえば、イベントハンドラを共有するコンポーネントのTagプロパティをそれぞれ違う値にしておけば、イベントハンドラでTComponent(Sender).Tagとすることで呼び出したコンポーネントを区別できます。

もっと直接的にコンポーネントの名前やクラス名（種類）を使うこともできます。次のイベントハンドラは、コンポーネントのクラス名と名前（キャプションではない）をフォームのタイトルバーに表示します。

```
procedure TForm1.BtnClick(Sender: TObject);
begin
  Caption := Sender.ClassName + ' - ' + TComponent(Sender).Name;
end;
```

次に、実行時のコントロールの生成ですが、Delphiでは設計時に配置するコンポーネントとまったく同じ形で実行時にコンポーネントを生成できます。フォーム上にボタンを配置し、そのボタンのイベントハンドラとして新しいボタンを生成するプログラムを以下に示します。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewButton: TButton;
  n: Integer;
begin
  { 作成済みのコンポーネントを調べて、使っていない番号を取得する }
  for n := 0 to ComponentCount - 1 do
    if FindComponent('NewButton_' + IntToStr(n)) = nil then
      Break;
  { Button コンポーネントの生成 }
  NewButton := TButton.Create(Self);
  with NewButton do
  begin
    { 以下、プロパティの設定 }
    SetBounds(20, n*40, 200, 30);              { 大きさ }
    Name := 'NewButton_' + IntToStr(n);      { コンポーネント名 }
    Caption := 'New Button #' + IntToStr(n);{ キャプション }
    OnClick := Button1Click;                   { イベントハンドラ }
    Parent := Self;                            { 親コンポーネント }
  end;
end;
```

FindComponentは、フォーム上に配置されたすべてのコンポーネントから与えられた名前のコンポーネントを見つけ出すメソッドです。フォーム上に配置されたコンポーネントは、Componentsという配列プロパティに保持されているので、この配列に保持されているコンポーネントのNameプロパティを調べているわけです。ここでは、'NewButton_'+数値という名前のコンポーネントが見つからなくなるまで繰返して、新しいコンポーネントに割り当てる番号を探しています（別の整数変数を使えば、もっと簡単に処理できるでしょう）。

NewButton := TButton.Create(Self);という記述で、新しいボタンオブジェクトを生成しています。ここでは、少なくともParentプロパティを設定しておく必要があります。フォーム上に配置するならParent := Self;としますが、パネルやグループボックス上にボタンを配置するなら、Parent := Panel1;などとパネルやグループボックスのコンポーネント名を指定します。通常は、大きさやキャプションなども指定します。また、前述のFindComponentで調べるためコンポーネントの名前（Nameプロパティ）を'NewButton_'+数値にしておきます。

新しく生成したボタンのOnClickイベントハンドラにも、自分自身のイベントハンドラButton1Clickを指定しています。つまり、新しく作成したボタンをクリックしても、同じ動作（つまりコンポーネントの生成）になります。

コンポーネントを削除するのは、Freeメソッドを呼び出すだけです。次のようにすれば、番号nで指定したコンポーネントを削除できます。

```
var
  cp: TComponent;
begin
  cp := FindComponent('NewButton_' + IntToStr(n));
  if cp <> nil then
    cp.Free;
end;
```

コントロール配列全体をまとめて処理するのは、Delphiではコンポーネント名やキャプションなど他のプロパティを使います。たとえば、コンポーネントの名前を'GroupBtn_'+数値のような形で統一しておき、FindComponentを使えばこの形式の名前のコンポーネントをまとめて処理できます。GroupBtn_1からGroupBtn_5というボタンコンポーネントがあれば、次のように処理できます。

```
var
  i: Integer;
  cp: TComponent;
begin
  { GroupBtn_1 から GroupBtn_5 までの Button コンポーネントの }
  { キャプションをイタリック体にする }
  for i := 1 to 5 do
  begin
    cp := FindComponent('GroupBtn_' + IntToStr(i));
    if cp <> nil then
      TButton(cp).Font.Style := [fsItalic];
  end;
end;
```

CHAP6¥CTLARRY.DPR

Q. Visual BasicのフォームのAutoRedrawプロパティに相当するものはないでしょうか。

Visual BasicのAutoRedrawプロパティは、フォーム自身のイメージをビットマップデータとして保持し、再描画の手間を省くためのものです。また、再描画途中の経過を表示しなくなるので、アニメーションのように動的な描画でもチラつかなくなります。

DelphiのフォームにはAutoRedrawに相当するプロパティはありませんが、イメージはImageコンポーネントで処理できるので、これをフォームに配置します。フォームの大きさが変更されたときのために、ImageコンポーネントのAlignプロパティはalClientにします。

また、フォームのOnResizeイベントハンドラを次のように定義します。これは、フォームの大きさが広げられたときにイメージが持つビットマップの大きさも広げるためのものです。ただし、フォームが縮小化されても、ビットマップの範囲は縮小化されません。

```
procedure TForm1.FormResize(Sender: TObject);
begin
  if Image1.Picture.Bitmap.Width < ClientWidth then
    Image1.Picture.Bitmap.Width := ClientWidth;
  if Image1.Picture.Bitmap.Height < ClientHeight then
    Image1.Picture.Bitmap.Height := ClientHeight;
end;
```

さらに、フォームの背景を描画しないように背景を再描画するメッセージを処理します。

```
type
  TForm1 = class(TForm)
    ...
  protected
    procedure WMEraseBkgnd(var Msg: TWMEraseBkgnd);
              message WM_ERASEBKGND;
    ...
  end;

  ...

procedure TForm1.WMEraseBkgnd(var Msg: TWMEraseBkgnd);
begin
  Msg.Result := 1;
end;
```

こうしておけば、Image1.Canvasに対して描画することで、Visual BasicのAutoRedrawプロパティがTrueの場合のように機能します。

このフォームを使用するプログラム例として、フォームにTimerコンポーネントを配置し、OnTimerイベントハンドラを次のように定義してください。TimerのIntervalプロパ

ティは短い値(100程度)にしておきます。

```
procedure TForm1.Timer1Timer(Sender: TObject);
const
  Radius: Integer = 0;
  Dist: Integer = 3;
begin
  with Image1.Canvas do
  begin
    Pen.Width := 3;
    Pen.Color := clWhite;  { clWhite: 円を消去する色 }
    Ellipse(100-Radius, 100-Radius, 100+Radius, 100+Radius);
    Inc(Radius, Dist);
    if (0 < Dist) and (100 <= Radius)
        or (Dist < 0) and (Radius <= -100) then
      Dist := -Dist;
    Pen.Color := clRed;    { clRed: 円を描画する色 }
    Ellipse(100-Radius, 100-Radius, 100+Radius, 100+Radius);
  end;
end;
```

もし、イベントハンドラの先頭にあるImage1.CanvasをCanvasに、消去する色をclWhiteではなくSelf.Color（フォーム自身の色）にし、WM_ERASEBKGNDメッセージを処理しないようにするとフォームに対して円を描画することになります。この場合、処理は早くなりますが画面を書換える経過が目に見えてしまいます。

これは、フォームのCanvasが、スクリーンに対するデバイスコンテキストを指しているのに対し、ImageのCanvasはメモリ中のデバイスコンテキストを指しているためです。フォームのCanvasに描画することはスクリーンに直接描画することになりますが、ImageのCanvasはメモリ内の仮想ビットマップに描画するだけなので、描画途中の状態は目に見えません。このイベントハンドラを終了すると、ImageのCanvasに描画した内容がフォーム上に反映されます。また、メモリ内に描画結果が残っているので、フォームがアイコン化されたり他のフォームに隠された場合でも、ふたたびフォームが表示されれば描画した内容が復元されます。

Imageコンポーネントに対する描画は、イベントハンドラが終了するかUpdateメソッドが呼び出されるまで画面に反映されません。

CHAP6¥AREDRAW.DPR

Q. Visual Basicのジェネラルプロシージャのようなものは、どのように作成すればよいのでしょうか。[ファイル(F)｜新規作成(N)]でユニットを作成してもinterfaceの下に手続きを定義するとコンパイルエラーになり、implementationの下に定義するとコンパイルは成功しますが、他から呼び出せません。

外部から呼び出す手続きや関数は宣言だけを**interface**部に記述して、定義を**implementation**部に記述します。たとえば、次のように記述できます。

```
unit General;

interface

{ ビープ音を鳴らす手続きの宣言 }
procedure Beep;

implementation

{ ビープ音を鳴らす手続きの定義（実装） }
procedure Beep;
begin
  MessageBeep(MB_OK);
end;

end.
```

また、この手法さえ守っていれば汎用ルーチンを定義するために、わざわざ独立したユニットを作成する必要はありません。フォームユニットにも**interface**部と**implementation**部がありますが、同じように宣言と定義を記述して他のユニットから呼び出せる手続きや関数を作成できます。

さらに、フォームに関連付けられているメソッドや変数も別のフォームから呼び出せます。このとき、外部から参照するためのメソッドや変数は**public**（{ public宣言 }）という場所に記述します。

次のプログラム例は、フォームにIncrementというメソッド（手続き）を追加して、外部から呼び出せるようにしたものです。他のフォームからForm2.Increment;とすれば、このメソッドが呼び出されてForm2のキャプションが更新されます（呼び出すフォームユニットの**uses**節にJunk2を追加しておく必要があります）。

```
type
  TForm2 = class(TForm)
    Label1: TLabel;
  public
    procedure Increment;
  end;
```

```
  ...
procedure TForm2.Increment;
var
  n: Integer;
begin
  { フォームのキャプションを数値に変換する }
  try
    n := StrToInt(Caption);
  except
    { 変換エラーが発生したら 0 にする }
    on E:EConvertError do n := 0;
  end;
  { 1 だけ増加して、キャプションを設定し直す }
  Inc(n);
  Caption := IntToStr(n);
end;

end.
```

CHAP6¥USEBEEP.DPR

Q. Visual Basicのライン（直線）コントロールに対応するものはないでしょうか。

基本的に、Delphiのコンポーネントはすべて長方形の領域として表現されます。このため、Visual Basicのラインのようなコンポーネントは実現しにくいのです。単に描画する目的だけであれば、PaintBoxコンポーネントを使ってOnPaintに直線の描画プログラムを記述できます。

付録CDには、長方形の対角線にラインを描画するLineコンポーネントを収録してあります。Lineコンポーネントは左上から右下、あるいは右上から左下へのラインを描画します。なお、線の幅が1より大きい場合、枠からはみ出す部分は描画されません。

CHAP6¥USELINE.DPR（Lineコンポーネントを使用）

Q. Visual BasicのFor文では、Stepで制御変数の増分を指定できましたが、Delphiではできないのでしょうか。

Delphiの**for**文では、制御変数を1ずつ増加させる(**to**)か減少させる(**downto**)かのどちらかとなります。Visual Basicの次のプログラムは、

```
For I = 0 to 20 step 4
  Sum = Sum + I
Next I
```

Delphiでは次のようになります。

```
for I := 0 to 5 do
  Sum := Sum + I * 4;
```

あるいは**while**文を使って次のようにも記述できます。

```
I := 0;
while I <= 20 do
begin
  Sum := Sum + I;
  Inc(I, 4);
end;
```

Object Pascalの**for**文は、Visual Basicよりも厳しい条件があります。主な条件は以下のとおりです。

・制御変数は、順序型（整数型、論理型、列挙型など）でなければならない。
・制御変数は、**for**文のブロックに属するローカル変数でなければならない。

Visual Basicでは、繰返しの途中で制御変数に終了値よりも大きい値（Stepが負のときは小さい値）を指定して繰返しを終了させることができますが、Object Pascalでは制御変数には変更できません。繰返し文を中断させるためには、Break文を使ってください。これは、Visual BasicのExit For文と同じように機能します。

CHAP6¥FORSTEP.DPR

Q. Visual BasicのフォームのScaleLeftやScaleWidthに対応するプロパティはないのですか。

Delphiでは、フォームやPaintBoxのキャンバスへの描画は常にピクセル単位で扱われています。このため、スケールを指定して描画するためには、フォームの領域と描画したい領域を対応させる計算が必要になります。

付録CDには、疑似的にスケール機能を使えるようにするためのCanvasScaleコンポーネントが収録されています。このコンポーネントは、Canvas（TCanvas型）をもとに指定されたスケールで描画できるようにするため、Delphiのオブジェクト型（**object**）を使っています。

コンポーネントを使うためには、CanvasScaleをフォームに配置し、ScaleLeft、ScaleTop、ScaleWidth、ScaleHeightプロパティを指定します。便宜上、ScaleRight、ScaleBottomも用意されていますが、これらの変更はScaleWidth、ScaleHeightに反映されます。

スケールを使った描画は、CanvasScale1[Canvas]に対するメソッドを使います。このとき**with**文を使えば、いちいちCanvasScale1を指定する必要はなくなります。

たとえば、ScaleLeft、ScaleTop、ScaleWidth、ScaleHeightが-1、-1、2、2のとき次のプログラムは、フォームの対角線を描画します。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with CanvasScale1[Canvas, ClientRect] do
  begin
    MoveTo(-1, -1);
    LineTo(1, 1);
    MoveTo(-1, 1);
    LineTo(1, -1);
  end;
end;
```

CHAP6¥USESCALE.DPR（CanvasScaleコンポーネントを使用）

Q. Visual Basicでは、Chrに2バイト値を代入すると漢字（2バイト文字）が返されましたが、Delphiではどうすればよいでしょうか。

Visual Basicでは16進を表現するために&Hxxxxと記述しますが、Object Pascalでは先頭に$を付けて$xxxxと記述します。またObject PascalのChrは、Char型（1バイト型）を返します。たとえば、全角の「B」（文字コード8261H）という文字を得るためにChr($8261)としても、下位バイトだけが有効になり半角の「a」が返されます。

2バイト文字は、1バイト文字を連結したものなのでChr($82)+Chr($61)とすれば全角の「B」になります。また、文字コードが決まっている場合は文字コードの前に#を付けて文字として使うことができます。たとえば、#$82 + #$61とすればよいわけです。さらに、#による文字定数どうしや文字定数とシングルクォートによる文字列は+を使わず連続して記述することもできるので、#$82#$61と記述することもできます。

ただし、変数を使って、#($61+n)とすることはできません。また、2つの文字列定数を連結するときは+が必要です。たとえば、'ABC''DEF'と記述するとABC'DEFという意味になります。これは、Object Pascalでは連続したシングルクォート('')は一つのシングルクォート(')をあらわすという意味があるためです。'ABC' 'DEF'のようにスペースを空けるとエラーになります。

Q. Visual BasicでファイルにデータをDelphiで利用したいのですが、どうすればよいでしょうか。

Visual BasicでPrint #などを使ってデータをテキスト文字列として出力している場合、DelphiではTextFile型の変数とAssignFile、Resetなどの手続きを使って読み込むことができます。テキスト文字列に対応するデータ形式の違いについては、プログラミングで対応する必要があります。

しかし、Put #1などでバイナリデータのまま出力されたものは、両者のデータ表現の違いについて注意する必要があります。これらの違いとさまざまなファイル入出力形式について具体的なプログラムとともに説明します。

Visual Basicの主なデータ型に対するObject Pascalのデータ型は、以下のとおりです。

データ型	Visual Basic	Delphi	備考
バイト型	Byte	Byte	
ブール型	Boolean	WordBool	
短精度整数型	Integer	Smallint	
長精度整数型	Long	Longint	
単精度実数型	Single	Single	
倍精度実数型	Double	Double	
通貨型	Currency	Currency	
日付型	Date	TDateTime	
可変長文字列型	String	String	変換が必要
固定長文字列型	String * n	WideCharの配列	
バリアント型	Variant	Variant	
オブジェクト型	Object	Variant	
配列	(Low To High)	array	次元の評価順が逆
ユーザー定義型	Type	record	

Delphi 3では、多くの型がそのまま対応して使えます。しかし、Visual BasicのIntegerは16ビットであるのに対し、Delphi 3のIntegerが32ビットであるなどの違いがあります。また、文字列型のファイルへの記録についても違いがあります。

Visual Basicでデータを保存するプログラム例と、Delphiでデータを読み込むプログラム例を示します。

まず、数値型については、型の対応に注意していればVisual Basicで記録された値は、問題なく読み込めるでしょう。逆に、Visual BasicにはDelphi 3のBoolean、Extended、Compなどに対応する型はありません。

TFileStreamの使い方については、オンラインヘルプの「TFileStream」を参照してください。

◆Visual Basicのプログラム

```
Private Sub Command1_Click()
  Dim Y As Byte
  Dim B As Boolean
  Dim I As Integer
  Dim L As Long
  Dim S As Single
  Dim D As Double
  Dim C As Currency
  Dim T As Date

  ' 定義した変数に適当な値を代入しておく
  Y = 100
  B = True
  I = &H1234
  L = 12345678
  S = 12.345
  D = 12345.678
  C = 1234567890.1234
  T = #5/30/64 2:08:00 PM#
  ' バイナリファイルとしてオープンする
  Open "TEST1.DAT" For Binary As #1
  ' 変数を順に記録する
  Put #1, , Y
  Put #1, , B
  Put #1, , I
  Put #1, , L
  Put #1, , S
  Put #1, , D
  Put #1, , C
  Put #1, , T
  Close #1
End Sub
```

◆Delphiのプログラム

```
function BoolToStr(B: Boolean): string;
begin
  if B then
    Result := 'True'
  else
    Result := 'False';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Y: Byte;
  B: WordBool;
  I: Smallint;
  L: Longint;
  S: Single;
  D: Double;
  C: Currency;
  T: TDateTime;
  F: TFileStream;
```

```
begin
  { ファイルストリームを作成し、読み込み専用ファイルとしてオープン }
  F := TFileStream.Create('TEST1.DAT', fmOpenRead);
  F.Read(Y, SizeOf(Y));
  F.Read(B, SizeOf(B));
  F.Read(I, SizeOf(I));
  F.Read(L, SizeOf(L));
  F.Read(S, SizeOf(S));
  F.Read(D, SizeOf(D));
  F.Read(C, SizeOf(C));
  F.Read(T, SizeOf(T));
  F.Free;
  { 以下のプログラムのために、ListBox コンポーネントを配置しておく }
  { ListBox1 の項目として、読み込んだデータを追加する }
  with ListBox1.Items do
  begin
    Clear;
    Add(IntToHex(Y, 4));
    Add(BoolToStr(B));
    Add(IntToHex(I, 4));
    Add(IntToStr(L));
    Add(FloatToStr(S));
    Add(FloatToStr(D));
    Add(CurrToStr(C));
    Add(DateTimeToStr(T));
  end;
end;
```

可変長文字列型は、2バイトの文字列長と文字列データとして記録されます。固定長文字列は、指定された長さの文字列データが記録されます。Delphiの長い文字列は、任意の長さの文字列を扱えるため、対応は比較的容易です。可変長文字列では読み込んだ文字列長、固定長文字列では指定した文字列長を**string**型変数に指定して、文字列データを読み込めばよいのです。

ただし、文字列データの読み込みはVisual Basicデータの読み込みに関わらず少しやっかいな面があります。Delphi 3では長い文字列を扱えるようにしたため、TFileStreamのRead、Writeメソッドには**string**型をそのまま指定できません。必ず、PChar(str)^のような型変換が必要になります。

TFileStreamのRead、Writeメソッドには、**string**型の変数をそのまま指定してはいけません。

Visual Basicでデータを保存するプログラム例と、Delphiでデータを読み込むプログラム例を示します。また、Delphiのプログラムだけでも使えるよう文字列型データを書き込む手続きも示します。

◆Visual Basicのプログラム

```
Private Sub Command2_Click()
  Dim VStr As String
  Dim FStr As String * 20

  ' ランダムファイルとしてオープンする（レコードサイズ：32バイト）
  Open "TEST2.DAT" For Random As #1 Len = 32
  ' 変数を順に記録する
  VStr = "Variable Size"
  Put #1, 1, VStr            ' 第 1 レコードに可変長文字列を記録
  FStr = "Fixed Size"
  Put #1, 4, FStr            ' 第 4 レコードに固定長文字列を記録
  VStr = "Another String"
  Put #1, 3, VStr            ' 第 3 レコードに可変長文字列を記録
  FStr = "Last String"
  Put #1, 2, FStr            ' 第 2 レコードに固定長文字列を記録
  Close #1
End Sub
```

◆Delphiのプログラム

```
{ ファイルストリームから、可変長文字列を読み込む }
function ReadVarString(F: TFileStream): string;
var
  VStrSize: Word;                          { 可変長文字列の長さ }
begin
  F.Read(VStrSize, SizeOf(VStrSize));  { 可変長文字列の長さを読み込む }
  SetLength(Result, VStrSize);
  F.Read(PChar(Result)^, VStrSize);
end;

{ ファイルストリームから、固定長文字列を読み込む }
function ReadFixedString(F: TFileStream; Size: Word): string;
begin
  SetLength(Result, Size);
  F.Read(PChar(Result)^, Size);
end;

{ ファイルストリームへ、可変長文字列として書き込む }
procedure WriteVarString(F: TFileStream; const Value: string);
var
  VStrSize: Word;                          { 可変長文字列の長さ }
begin
  VStrSize := Length(Value);
  F.Write(VStrSize, SizeOf(VStrSize)); { 可変長文字列の長さを読み込む }
  F.Write(PChar(Value)^, VStrSize);
end;

{ ファイルストリームへ、固定長文字列として指定された長さを書き込む }
procedure WriteFixedString(F: TFileStream; const V: string; Size:
Word);
begin
  F.Write(Size, SizeOf(Size));
  F.Write(PChar(V)^, Size);
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
const
  RecordSize = 32;                { Visual Basic の"Len ="に対応する値 }
var
  F: TFileStream;
begin
  { 読み込んだデータを表示するために ListBox コンポーネントを使う }
  ListBox1.Items.Clear;
  { ファイルストリームを作成し、読み込み専用ファイルとしてオープン }
  F := TFileStream.Create('TEST2.DAT', fmOpenRead);
  { ListBox1 の項目として、読み込んだデータを随時追加する }
  F.Seek(RecordSize * 0, 0);              { 第 1 レコードへ移動 }
  ListBox1.Items.Add(ReadVarString(F));
  F.Seek(RecordSize * 1, 0);              { 第 2 レコードへ移動 }
  ListBox1.Items.Add(ReadFixedString(F, 20));
  F.Seek(RecordSize * 2, 0);              { 第 3 レコードへ移動 }
  ListBox1.Items.Add(ReadVarString(F));
  F.Seek(RecordSize * 3, 0);              { 第 4 レコードへ移動 }
  ListBox1.Items.Add(ReadFixedString(F, 20));
  F.Free;
end;
```

バリアント型は、まず保持しているデータの型を表わす2バイトのデータが書き込まれ、続いて保持しているデータそのものが記録されます。バリアント型には、表に示したほとんどの型に加え、Empty、Nullを保持できますが、EmptyとNullには保持するデータはありません。

Delphi 3にもバリアント型はありますが、ファイルとの読み書きをサポートする手続きは用意されていません。このため、バリアント型のデータを処理するためのプログラムを作成する必要があります。バリアント型の内部構造に対応するレコード型としてTVarDataがありますから、これを使えばよいでしょう。

Visual Basicでデータを保存するプログラム例と、Delphiでデータを読み込むプログラム例を示します。また、Delphiのプログラムだけでも使えるようバリアント型データを書き込む手続きも示します。なお、バリアント配列（バリアント型に任意次元の配列を格納するもの）については対応していません。

◆Visual Basicのプログラム

```
Private Sub Command3_Click()
  Dim V As Variant
  Open "TEST3.DAT" For Binary As #1
  Put #1, , V       ' Empty の書き込み
  V = Null
  Put #1, , V       ' Null の書き込み
  V = 123
  Put #1, , V       ' Integer の書き込み
  V = 123456789
  Put #1, , V       ' Long の書き込み
  V = 12.345!
  Put #1, , V       ' Single の書き込み
  V = 12345.6789
```

```
  Put #1, , V       ' Double の書き込み
  V = 1234567.89@
  Put #1, , V       ' Currency の書き込み
  V = Now
  Put #1, , V       ' 日付時間型の書き込み
  V = "STRING"
  Put #1, , V       ' 文字列の書き込み
  Close #1
End Sub
```

◆Delphiのプログラム

```
{ ファイルストリームから、バリアント型データを読み込む }
function ReadVariant(F: TFileStream): Variant;
var
  VData: TVarData;
  Len: Word;
  Str: string;
begin
  with VData do
  begin
    F.Read(VType, SizeOf(VType));
    case VType of
    varEmpty: ;
    varNull: Result := Null;
    varSmallint:
      begin
        F.Read(VSmallint, SizeOf(VSmallint));
        Result := VSmallint;
      end;
    varInteger:
      begin
        F.Read(VInteger, SizeOf(VInteger));
        Result := VInteger;
      end;
    varSingle:
      begin
        F.Read(VSingle, SizeOf(VSingle));
        Result := VSingle;
      end;
    varDouble:
      begin
        F.Read(VDouble, SizeOf(VDouble));
        Result := VDouble;
      end;
    varCurrency:
      begin
        F.Read(VCurrency, SizeOf(VCurrency));
        Result := VCurrency;
      end;
    varDate:
      begin
        F.Read(VDate, SizeOf(VDate));
        Result := TDateTime(VDate);
      end;
```

```
    varOleStr:
      begin
        F.Read(Len, SizeOf(Len));
        SetLength(Str, Len);
        F.Read(PChar(Str)^, Len);
        Result := Str;
      end;
    varBoolean:
      begin
        F.Read(VBoolean, SizeOf(VBoolean));
        Result := VBoolean;
      end;
    varByte:
      begin
        F.Read(VByte, SizeOf(VByte));
        Result := VByte;
      end;
    else { varString, varDispatch, varError, varUnknown }
      raise EVariantError.Create('Unhandled variant in FileStream');
    end;
  end;
end;

{ ファイルストリームへ、バリアント型データを書き込む }
procedure WriteVariant(F: TFileStream; const Value: Variant);
var
  VData: TVarData;
  Len: Word;
  Str: string;
begin
  with VData do
  begin
    VType := VarType(Value);
    { varString は Visual Basic でサポートされていない }
    if VType = varString then
      VType := varOleStr;
    F.Write(VType, SizeOf(VType));
    case VType of
    varEmpty, varNull: ;
    varSmallint:
      begin
        VSmallInt := Value;
        F.Write(VSmallint, SizeOf(VSmallint));
      end;
    varInteger:
      begin
        VInteger := Value;
        F.Write(VInteger, SizeOf(VInteger));
      end;
    varSingle:
      begin
        VSingle := Value;
        F.Write(VSingle, SizeOf(VSingle));
      end;
    varDouble:
      begin
        VDouble := Value;
        F.Write(VDouble, SizeOf(VDouble));
      end;
```

```
    varCurrency:
      begin
        VCurrency := Value;
        F.Write(VCurrency, SizeOf(VCurrency));
      end;
    varDate:
      begin
        VDate := Value;
        F.Write(VDate, SizeOf(VDate));
      end;
    varOleStr:
      begin
        Str := Value;
        Len := Length(Str);
        F.Write(Len, SizeOf(Len));
        F.Write(PChar(Str)^, Len);
      end;
    varBoolean:
      begin
        VBoolean := Value;
        F.Write(VBoolean, SizeOf(VBoolean));
      end;
    varByte:
      begin
        VByte := Value;
        F.Write(VByte, SizeOf(VByte));
      end;
    else { varDispatch, varError, varUnknown }
      raise EVariantError.Create('Unhandled variant in FileStream');
    end;
  end;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  V: Variant;
  I: Integer;
  F: TFileStream;
begin
  { 読み込んだデータを表示するために ListBox コンポーネントを使う }
  ListBox1.Items.Clear;
  { ファイルストリームを作成し、読み込み専用ファイルとしてオープン }
  F := TFileStream.Create('TEST3.DAT', fmOpenRead);
  { ListBox1 の項目として、読み込んだデータを随時追加する }
  for I := 0 to 8 do
  begin
    V := ReadVariant(F);
    ListBox1.Items.Add(VarToStr(V));
  end;
  F.Free;
end;
```

配列はVisual Basicではそのまま保存することはできませんが、Typeを使ったユーザー定義型の一部として保存することはできます。個々の要素は上記に記した通りで、1次元配列では要素の範囲を合わせておけばそのまま読み込めます。ただし、可変長文字列型については、上記の処理を繰り返す必要があります。多次元配列については、Visual Basic

では要素の並びが左側の次元が優先して更新されるのに対し、Delphiでは右側の次元が優先して更新されます。また、Delphi 3では高速化のためにユーザー定義型の各フィールドを4バイト境界位置に並べようとします。これを避けるためには、**record**の定義に**packed**を指定します。

配列とユーザー定義型を組み合わせて、Visual Basicでデータを保存するプログラム例と、Delphiでデータを読み込むプログラム例を示します。

◆Visual Basicのプログラム（型の定義）

```
Type NewType
  Matrix(1 To 3, 2 To 5) As Integer
  StrA As String * 10
  IntA As Long
  CurA As Currency
End Type
```

◆Visual Basicのプログラム

```
Private Sub Command4_Click()
  Dim X As NewType
  Dim I, J As Integer

  For I = 1 To 3
    For J = 2 To 5
      X.Matrix(I, J) = I * 10 + J
    Next
  Next
  X.StrA = "ABCDEFG"
  X.IntA = 1234
  X.CurA = 12345@
  Open "TEST4.DAT" For Binary As #1
  Put #1, , X
  Close #1
End Sub
```

◆Delphiのプログラム

```
type
  NewType = packed record
    Matrix: array [2..5, 1..3] of Smallint; { 次元の順序を逆にする }
    StrA: array [0..9] of Char;             { 0 から始まる文字配列 }
    IntA: Longint;
    CurA: Currency;
  end;

procedure TForm1.Button4Click(Sender: TObject);
var
  X: NewType;
  I, J: Integer;
  Temp: array [0..10] of Char;
```

```
  F: TFileStream;
begin
  { ファイルストリームを作成し、読み込み専用ファイルとしてオープン }
  F := TFileStream.Create('TEST4.DAT', fmOpenRead);
  { 一つのレコードを読み込む }
  F.Read(X, SizeOf(X));
  F.Free;

  { リストボックスに読み込んだデータを項目として追加する }
  ListBox1.Items.Clear;
  for I := 1 to 3 do
    for J := 2 to 5 do
      ListBox1.Items.Add(IntToStr(X.Matrix[J, I]));
  { 文字列データはヌルで終わる文字列にしてから、Pascal形式に変換する }
  StrLCopy(Temp, X.StrA, 10);
  Temp[10] := #0;
  ListBox1.Items.Add(StrPas(Temp));
  ListBox1.Items.Add(IntToStr(X.IntA));
  ListBox1.Items.Add(CurrToStr(X.CurA));
end;
```

CHAP6¥RDVBDAT.DPR
CHAP6¥MKVBDAT.VBP

Q. Visual BasicのプログラムからDelphiで作成したDLLを呼び出したいのですが、値の受渡しはどのようにすればよいでしょうか。

DelphiとVisual Basicのプログラムで、お互いに呼び出すためにはOLEオートメーションを使うのが簡単です。OLEオートメーションを使えば、呼び出し形式やデータの受渡しについて悩まされることなく、プログラムが提供するメソッドやプロパティを制御できます。OLEオートメーションについては、それぞれのマニュアル（Delphiでは『ユーザーズガイド』第15章）を参照してください。

しかし、OLEオートメーションは呼び出しのためにオーバーヘッドがかかるため、頻繁に互いを呼びだし合う場合は、速度が低下する原因にもなりかねません。この場合は、DLLでエクスポート関数を定義し、利用することもできます（なお、Visual Basicのプログラムではエクスポート関数を定義できません）。エクスポート関数に値を渡す場合も、ファイルからデータを読み込む場合とほとんど同じです。型の対応については前項目を参照してください。

Delphi側では、エクスポートしたい関数を**stdcall**を使って宣言します。**stdcall**は、Win32で一般的に使われている呼びだし形式です。Win16ではPASCAL形式が使われていたため、Delphiでは特に呼び出し形式を指定する必要はありませんでしたが、Win32では**stdcall**を付けないと引数を正しく評価できなくなりますので注意してください。

Visual Basicのプログラムの中でDLLのエクスポート関数を使うための宣言(Declare)文では、引数にByValを付けます。もし、Visual Basicの宣言でByValを付けない場合は、Object Pascalのエクスポート関数の引数に**var**をつけて変数引数として宣言します。

文字列(String)については必ずByValを付けます。文字列引数をByVal付きで宣言することで、引数をPCharで処理できるヌルで終わる文字列として受け取ることができます。Delphiの文字列型とVisual Basicの文字列型は、内部処理が異なるため、ByValを付けない文字列引数はDelphi側で正しく処理できません。

バリアント型は、ByValなしで宣言してください。また、バリアント型が文字列を保持している場合は、エクスポート関数では文字列を正しく処理できません。

以下に、Delphiで作成するDLLのプログラム例と、Visual Basicで作成するDLLを利用するプログラム例を示します。Delphiのプログラムは、プロジェクトソース(.DPR)のみで構成されており、ユニットファイルは使っていません。

◆Delphiのプログラム（VBVALUE.DPR）

```
library VbValue;

uses
  Windows, SysUtils, Classes;
```

```
{ すべての型に対応し、型名と値を表示する内部関数 }
procedure DisplayValue(TypeName: string; Value: Variant);
var
  Msg: string;
begin
  Msg := TypeName + ': ' + VarToStr(Value);
  MessageBox(0, PChar(Msg), 'VBVALUE', MB_OK);
end;

{ 以下のエクスポート関数は、それぞれの型に対応する }
procedure DisplaySmallint(Value: Smallint); stdcall;
begin
  DisplayValue('Smallint', Value);
end;

procedure DisplayLongint(Value: Integer); stdcall;
begin
  DisplayValue('Longint', Value);
end;

procedure DisplaySingle(Value: Single); stdcall;
begin
  DisplayValue('Single', Value);
end;

procedure DisplayDouble(Value: Double); stdcall;
begin
  DisplayValue('Double', Value);
end;

procedure DisplayCurrency(Value: Currency); stdcall;
begin
  DisplayValue('Currency', Value);
end;

procedure DisplayString(Value: PChar); stdcall;
begin
  DisplayValue('String', StrPas(Value));
end;

procedure DisplayVariant(var Value: Variant); stdcall;
begin
  DisplayValue('Variant', Value);
end;

exports DisplaySmallint, DisplayLongint, DisplaySingle, DisplayDouble,
        DisplayCurrency, DisplayString, DisplayVariant;

begin
end.
```

◆Visual Basicのプログラム（宣言）

```
Private Declare Sub DisplaySmallint Lib "VBVALUE.DLL" (ByVal Value As
Integer)
Private Declare Sub DisplayLongint Lib "VBVALUE.DLL" (ByVal Value As
Long)
```

```
Private Declare Sub DisplaySingle Lib "VBVALUE.DLL" (ByVal Value As
Single)
Private Declare Sub DisplayDouble Lib "VBVALUE.DLL" (ByVal Value As
Double)
Private Declare Sub DisplayCurrency Lib "VBVALUE.DLL" (ByVal Value As
Currency)
Private Declare Sub DisplayString Lib "VBVALUE.DLL" (ByVal Msg As
String)
Private Declare Sub DisplayVariant Lib "VBVALUE.DLL" (Value As Variant)
```

◆Visual Basic のプログラム（呼び出し）

```
Private Sub Command1_Click()
  DisplaySmallint (12345)
  DisplayLongint (123456789)
  DisplaySingle (12.345!)
  DisplayDouble (12345.6789)
  DisplayCurrency (1234567.89@)
  DisplayString ("STRING")
  V = Now
  DisplayVariant (V)
  V = 123456789
  DisplayVariant (V)
  V = 12345.6789
  DisplayVariant (V)
End Sub
```

CHAP6¥VBVALUE.DPR
CHAP6¥PASSVAL.VBP

第7章

for C/C++プログラマ

本章では、C/C++の開発者がDelphiを使う際の質問について取り上げています。また、C/C++とDelphiのプログラムの併用についても紹介しています。

Q. C/C++のような条件コンパイルを使うことはできますか。

Delphiには、条件コンパイルのためにC/C++の#defineに相当するものとして{$D symbol}というコンパイル指令が用意されています。これは、symbolに記述した識別子を定義するためのものです。#defineと違って、具体的に置き換える内容を指定することはできません。

定義したシンボルはC/C++の#ifdef、#ifndef、#else、#endifに相当する{$IFDEF symbol}、{$IFNDEF symbol}、{$ELSE}、{$ENDIF}を使って条件コンパイルのために使えます。また、Delphi 3では以下のシンボルがコンパイラによってあらかじめ定義されます。

シンボル名	意味
CPU386	コンパイル環境が80386以上を搭載している
WIN32	ターゲットがWin32アプリケーションである
VER100	コンパイラのバージョン（10.0）
CONSOLE	[コンソールアプリケーションの作成]がチェックされている

Q. C/C++のreturnは、Object Pascalではどのように記述すればよいのでしょうか。

Object Pascalでは、関数や手続きから直ちに戻るためにExit;という文を使います。ただし、関数(function)の場合はC/C++のreturnのようにExitで戻り値を指定するのではありません。戻り値は、関数が終わるまで（またはExitで抜け出すまで）に関数名かResultに代入します。

古典的なPascalの構文では、関数名に値を代入することで戻り値を設定します。たとえば、次のようになります。

```
function Multiply(a, b: Double): Double;
begin
  Multiply := a * b;
end;
```

しかし、代入文（:=）の左側以外で関数名を使うと再帰呼び出しの意味となるため、関数の中でも計算結果を利用したい場合に不便が生じます。

```
var
  Counter: Integer;

{ 間違ったプログラミング例 }
function Foo: Boolean;
begin
  Foo := (Counter <> 0);
  if Foo then                    { 比較した結果ではなく、新たに関数 Foo を }
    ShowMessage('Foo <> 0'); { 呼びだそうとして、無限ループに陥る }
end;
```

このような場合、従来は次のようにローカル変数を定義して回避していました。

```
function Foo: Boolean;
var
  Flag: Boolean;
begin
  Flag := (Counter <> 0);      { 比較した結果を、一時的なローカル変数に }
  if Flag then                 { 代入するので、無限ループにはならない }
    ShowMessage('Foo <> 0');
  Foo := Flag;                 { 結果を関数名に代入する }
end;
```

しかし、Object PascalではResultという予約語が拡張されており、より使いやすくなっています。Resultは、前述のFlagのようなローカル変数と同じように扱えますが、関数名に代入するのと同じように関数の戻り値をあらわします。

```
function Foo: Boolean;
begin
  Result := (Counter <> 0);  { 比較した結果を Result に代入する }
  if Result then             { Result は関数呼び出しにはならない }
    ShowMessage('Foo <> 0');
end;                         { 関数名に代入し直す必要もない }
```

なお、C/C++で繰返し文の次のステップに進むためのcontinueや繰返し文を中断するbreakに相当するものとして、Object PascalにもContinueやBreakが組み込み手続きとしてサポートされています。

CHAP7¥FACTOR.DPR

Q. C/C++の演算子に対応するObject Pascalの演算子には、どのようなものがありますか。

Object Pascalには、C/C++のすべての演算子に対応するものがあるわけではありません。逆に、Object Pascalだけで使える演算子もあります。

たとえば、C/C++では代入(=)は演算子ですが、Object Pascalの代入(:=)は文です。このため、代入結果をさらに別の変数に代入するということはできません。このような注意は必要ですが、演算子や多くの組み込み手続きがC/C++の演算子の代わりに利用できます。

以下に、C/C++の演算子に対応するObject Pascalの演算子や組み込み手続きを示します。組み込み手続きはIncやDecのように値を返さない場合もあります。また、演算子の優先順位は、必ずしもC/C++と同等ではありません。

演算子の優先順位は、オンラインヘルプの「演算子/優先順位/式」を参照してください。

演算子	C/C++	Object Pascal
論理否定	!	not
ビット反転	~	not
単項プラス	+	+
単項マイナス	-	-
インクリメント	++	Inc
デクリメント	--	Dec
アドレス取得	&	@
ポインタ	*	^型名（定義）、ポインタ^（参照）
サイズ	sizeof	SizeOf
メモリ確保	new	New、Createメソッド
メモリ解放	delete	Dispose、Freeメソッド
メンバ参照	.*、->*	（※189ページ参照）
乗算	*	*
除算	/	div（整数の場合）、/（実数の場合）
剰余	%	mod
加算	+	+
減算	-	-
左シフト	<<	shl
右シフト	>>	shr
比較	<	<
	<=	<=
	>	>
	>=	>=
	==	=
	!=	<>
ビットAND	&	and
ビットOR	|	or
ビットXOR	^	xor

論理AND	&&	and
論理OR	||	or
三項演算子	?:	(if～then～else～)
代入	=	:=
加算代入	+=	Inc
減算代入	-=	Dec
他の演算代入	*=、/=、%=、&=、^=、|=、<<=、>>=	

Q. C/C++の共用体(union)は、Object Pascalではどのように定義すればよいのでしょうか。

C/C++の構造体はObject Pascalでレコード型（**record**）を使いますが、これと同じように共用体の代わりにもレコード型を使います。たとえば、長方形領域を表わすためのTRect型は、次のように定義されています。

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

ここで、**case** Integer **of**は、整数型で評価方法を分けるという意味です。0の場合には、Left、Top、Right、Bottomという整数型でアクセスでき、1の場合にはTopLeft、BottomRightというPoint型でアクセスできます。実際にアクセスするときは、0や1といった数値を意識する必要はありません。TRect型のRectという変数があれば、Rect.LeftやRect.BottomRightとしてアクセスできます。

caseの後ろには、通常のプログラム中の構文と同じように順序型であればどんな型でも使えます。たとえば、Integerの代わりにCharやBooleanを使って、次のように定義することもできます。

```
TRect = record
  case Char of
    'A': (Left, Top, Right, Bottom: Integer);
    'B': (TopLeft, BottomRight: TPoint);
end;

TRect = record
  case Boolean of
    False: (Left, Top, Right, Bottom: Integer);
    True: (TopLeft, BottomRight: TPoint);
end;
```

Q. C/C++のデータ型とObject Pascalのデータ型にはどんな違いがありますか。

A．C/C++の主なデータ型に対するObject Pascalのデータ型は、以下のとおりです(C/C++は32ビットコンパイラとします)。

データ型	C/C++	Delphi	備考
符号無文字型	unsigned char	Byte	
符号付文字型	char、signed char	Char	整数型ではない
ワイド文字型	wchar_t	WideChar	
符号無整数型	short	Smallint	
	int	Integer	
	long int	Longint	
符号無整数型	unsigned short	Word	
	unsigned int	Cardinal	
	unsigned long	DWord	
単精度実数型	float	Single	
倍精度実数型	double	Double	
倍長精度実数型	long double	Extended	
列挙型	enum	(識別子,...)	
構造体	struct	record	
共用体	union	record	case～ofを使う
クラス	class	class/object	
ユーザー定義型	typedef	(type)	

Object Pascalでは、**class**で定義するクラスを使うときは、必ずCreateメソッドとFreeメソッドを使って、明示的にオブジェクトの生成と解放を行なう必要があります。

Q. C/C++におけるメンバへのポインタや参照(.*、->*)は、Object Pascalではどのようになっていますか。

メンバ参照について、C++とObject Pascalでは大きな違いがあります。C++のメンバへのポインタは、任意のオブジェクトに対して利用しますが、Object Pascalではメソッドへのポインタはオブジェクトへのポインタとともに管理されます。また、クラスのフィールドへのポインタというものはなく、オブジェクトのフィールドへのポインタ（通常のデータへのポインタと等価）のみがあります。

次のC++プログラムを考えます。

```
class TSimple {
public:
    int Number;              // データメンバ
    void Increment(void);    // メンバ関数
    void Decrement(void);    // メンバ関数
};

// メンバ関数の実装
void TSimple::Increment(void)
{
    Number++;
}

void TSimple::Decrement(void)
{
    Number--;
}

// TSimple 型のオブジェクトの定義
TSimple X;

typedef void (TSimple::*s_fptr)(void); // メンバ関数へのポインタ

int something(void)
{
  s_fptr fptr = &TSimple::Increment;// メンバ関数へのポインタを設定

  X.Number = 10;             // データメンバに値を代入
  (X.*fptr)();               // メンバ関数へのポインタを使った呼び出し
  return X.Number;
}
```

Xがポインタで定義されていれば、メンバ関数へのポインタを使った呼び出しは、(X->*fptr)();のようになります。このプログラムに対応するObject Pascalのプログラムは次のようになります。

```
type
  TSimple = class
    Number: Integer;        { フィールド }
    procedure Increment;    { メソッド }
    procedure Decrement;    { メソッド }
  end;

{ メソッドの実装（implementation部に記述する） }
procedure TSimple.Increment;
begin
  Inc(Number);
end;

procedure TSimple.Decrement;
begin
  Dec(Number);
end;

var
  { TSimple 型のオブジェクトの定義 }
  X: TSimple;

type
  s_fptr = procedure of object;  { メソッドへのポインタ }

function Something: Integer;
var
  fptr: s_fptr;
begin
  X := TSimple.Create;    { 明示的なオブジェクトの生成 }
  fptr := X.Increment;    { メソッドへのポインタを設定 }

  X.Number := 10;         { フィールドに値を代入 }
  fptr;                   { メソッドへのポインタを使った呼び出し }
  Result := X.Number;     { フィールドを戻り値に設定 }

  X.Free;                 { 明示的なオブジェクトの解放 }
end;
```

C++では、void (A::*fptr)(void);のように直接メンバ関数へのポインタ変数を定義できますが、Object Pascalではあらかじめ**procedure** [(引数リスト)] **of object**か**function** [(引数リスト)] **of object**という形式で型を定義する必要があります。このときに特定のクラス名は指定しません。

Object Pascalのプログラムで、メソッドへのポインタfptrを設定するときに「クラス名.メソッド名」ではなく「オブジェクト名.メソッド名」を代入していることに注意してください。メソッドへのポインタには、実際にはメソッドへのポインタとオブジェクトへのポインタの両方が格納されます。このため、メソッドを呼び出すときにもオブジェクトを指定する必要はなく、C++のメンバ参照に対応する演算子はObject Pascalには存在しません。

メソッドへのポインタは、イベントハンドラの型として使われていますが、イベントハンドラを呼び出すためにメソッドの対象となるオブジェクトを指定する必要はありません。他のフォームに割り当てられているイベントハンドラを呼び出すために、

Form2.OnClick(Form2);のように記述できますが、これはOnClickがForm2のフィールドとして定義されているためです。同じことをC++で処理しようとすると、(Form2.*Form2.OnClick)(Form2);と記述することになります。

逆に、メソッドへのポインタを設定するときにオブジェクトへのポインタが必要になるため、オブジェクトが生成されていないときにメソッドへのポインタを使うことはできません。たとえば、メインフォームのOnCreateイベントハンドラで、まだ作成されていない他のフォームのイベントハンドラを利用することはできません。

もし、対象となるオブジェクトやメソッドのどちらかだけを変更する場合は、TMethod型を使います。

```
procedure Another;
var
  A, B: TSimple;
  fptr: s_fptr;
begin
  A := TSimple.Create;          { オブジェクトの作成 }
  B := TSimple.Create;          { オブジェクトの作成 }

  fptr :=
  A.Increment;                  { メソッドへのポインタの設定 }
  fptr;                         {   A.Increment; の呼び出し }
  TMethod(fptr).Data := B;      { 対象オブジェクトの変更 }
  fptr;                         { B.Increment; の呼び出し }
  TMethod(fptr).Code := @TSimple.Decrement; { メソッドの変更 }
  fptr;                         { B.Decrement; の呼び出し }

  B.Free;                       { オブジェクトの解放 }

  A.Free;                       { オブジェクトの解放 }
end;
```

CHAP7¥METHODP.DPR

Q. Cのtanやpowなど、対応する数学関数が見つかりません。

Delphi 3 Professional以上には、Mathユニットにこれらの関数が用意されています。Uses節にMathを追加すれば、三角関数やべき乗に対応する関数を利用できます。Delphi 3 Standardでは、サポートされる数学関数は限られているため、対応する関数がない場合は他の関数を使って計算する必要があります。Cの主な数学関数に対する対応表を以下に示します。

関数の意味	C/C++	Mathユニット	代替式
逆余弦	acos(x)	ArcCos(x)	ArcTan(Sqrt(1 - Sqr(x) / x))
逆正弦	asin(x)	ArcSin(x)	ArcTan(x / Sqrt(1 - Sqr(x)))
逆正接	atan(x)	→	ArcTan(x)
逆正接2	atan2(x, y)	ArcTan2(y, x)	ArcTan(y / x)
余弦	cos(x)	→	Cos(x)
双曲線余弦	cosh(x)	Cosh(x)	(Exp(x) + Exp(-x)) / 2
指数(底e)	exp(x)	→	Exp(x)
切り捨て	floor(x)	→	Trunc(x)またはInt(x)
自然対数	log(x)	→	Ln(x)
常用対数	log10(x)	Log10(x)	Ln(x) / Ln(10)
指数	pow(x, y)	Power(x, y)	Exp(y * Ln(x))
指数(底10)	pow10(x)	Power(10, x)	Exp(10 * Ln(x))
正弦	sin(x)	→	Sin(x)
双曲線正弦	sinh(x)	Sinh(x)	(Exp(x) - Exp(-x)) / 2
平方根	sqrt(x)	→	Sqrt(x)
正接	tan(x)	Tan(x)	Sin(x) / Cos(x)
双曲線正接	tanh(x)	Tanh(x)	(Exp(x)-Exp(-x))/(Exp(x)+Exp(-x))

Q. **Cのprintf関数のように、書式指定付きで数値や文字列を表示することはできませんか。**

Object Pascalには、任意の引数をprintf関数のように書式指定付きで表示したり利用するための関数としてFormatなどの文字列形式ルーチンが用意されています。文字列形式ルーチンには、次の5種類があります。

FmtStr	書式化した文字列（string型）をResult引数に返す手続きです。
Format	書式化した文字列（string型）を戻り値とする関数です。
FormatBuf	書式化した文字列を指定したバッファに返し、文字数を返す関数です。
StrFmt	書式化した文字列（PChar型）をBuffer引数に返す関数です。
StrLFmt	書式化した文字列（PChar型）を上限付きのBuffer引数に返す関数です。

たとえば、Format('%d,%s', [123, 'ABC'])とすると'123,ABC'という文字列が得られます。文字列形式ルーチンには、Cのprintfと違ってカンマ区切り表示や通貨表示などがサポートされています。あまり使われない整数以外でのゼロサプライ表示（数値の右寄せ表示で余った桁に0を表示すること）などはサポートされていません。なお、ゼロサプライは'%05d'ではなく'%.5d'のように指定します。Currency型やComp型は浮動小数型として扱われますので、整数型を対象とする%dなどは使えないので注意してください。

なお、scanfに相当する手続きや関数はありません。

形式文字列の詳細は、オンラインヘルプの「形式文字列」を参照してください。

CHAP7¥DISPFMT.DPR

Q. C/C++のva_startやva_argを使った可変個引数に対応する手続きや関数は作成できますか。

Object Pascalではオープン配列という形式で、可変個の引数を受け取ることができます。オープン配列では、C/C++の可変個引数よりもずっと簡単かつ安全に引数を処理できます。

引数をオープン配列として宣言するためには、**array**の後ろの範囲指定を省略します。また、関数を呼び出すときは、引数列を[]で囲みます。オープン配列を使った簡単な関数を以下に示します。

```
function Total(const Params: array of Integer): Integer;
var
  i: Integer;
begin
  Result := 0;
  for I := Low(Params) to High(Params) do
    Inc(Result, Params[i]);
end;

  ...

{ Total を呼び出す例 }
var
  Sum: Integer;
begin
  Sum := Total([123, 456, 789]);
  ...
```

LowとHighは、指定された範囲の下限と上限を返す組み込み関数で、ここでは渡された引数の範囲を表わします。オープン配列の下限は常に0から始まるためLow(Params)は0に置き換えてもかまいません。C/C++のように引数を与えられた順に評価する必要はなく、範囲もわかっているため終了条件を与える必要もありません。

さらに、任意の型の引数を渡すために型保障のオープン配列というものが用意されています。このためには、オープン配列の型の代わりに**const**を指定して**array of const**とします。型保障付きのオープン配列を使った例を以下に示します。この場合は、関数を呼び出す際に任意の型の値を渡すことができます。また、型保障のオープン配列を受け取る関数は、引数をTVarRec型の配列とみなして処理できます。

型保障のオープン配列を使った関数の例を以下に示します。

```
function ToStr(const Params: array of const): string;
var
  i: Integer;
  s: string;
begin
  Result := '';
  for i := 0 to High(Params) do
  begin
    { 渡された値の型に応じて、適切な文字列に変換する }
    with Params[i] do
      case VType of
      vtInteger: s := IntToStr(VInteger);
      vtBoolean: if VBoolean then s := 'True' else s := 'False';
      vtChar: s := VChar;
      vtExtended: s := FloatToStr(VExtended^);
      vtString: s := VString^;
      vtPointer: s := Format('%p', [VPointer]);
      vtPChar: s := StrPas(VPChar);
      vtObject: s := VObject.ClassName;
      vtClass: s := 'class ' + VClass.ClassName;
      vtWideChar: s := '$' + IntToHex(Ord(VWideChar), 4);
      vtPWideChar: s := WideCharToString(VPWideChar);
      vtAnsiString: s := StrPas(VAnsiString);
      vtCurrency: s := Format('%m', [VCurrency]);
      vtVariant: s := 'Variant ' + IntToHex(VarType(VVariant^), 4);
      else s := '<<unknown>>';
      end;
    { 文字列を連結する }
    if i > 0 then
      Result := Result + ',';
    Result := Result + s;
  end;
end;

  ...

{ ToStr を呼び出す例 }
begin
  Edit1.Text := ToStr([123, 456.78, Edit1, 'string']);
  ...
```

TVarRec型の定義については、オンラインヘルプの「TVarRec 型」を参照してください。

任意の型を文字列として扱うためには、バリアント型を使うこともできます。前述と同様のプログラムは、バリアント型を使えば以下のように簡単に記述できます。

```
function ToStrV(const Params: array of Variant): string;
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Params) do
  begin
    if I > 0 then
      Result := Result + ', ';
    Result := Result + VarToStr(Params[I]);
  end;
end;
```

CHAP7¥OPENARR.DPR

Q. C++でnew型[要素数];とするように、可変長の動的配列をヒープメモリから確保するには、どうすればよいでしょうか。

Object Pascalでは、New/Deleteを使ってヒープメモリを扱う場合、配列の要素数はあらかじめ決めておく必要があります。しかし、ポインタとメモリ確保ルーチンを組み合わせて、任意の大きさのメモリを確保して利用できます。

まず、次のようにあらかじめ配列型と配列型へのポインタを定義しておきます。このとき、配列の範囲は想定される範囲の上限を設定しておいてください。これは、配列要素にアクセスする際、範囲チェック（{$R+}）でエラーにならないようにするためです。ここでは、Integer型で説明しますが、レコード型でも同様に処理できます。

```
type
  PIntArray = ^TIntArray;
  TIntArray = array of [0..100] of Integer;
```

配列へのポインタ型の変数を定義し、AllocMemを使ってメモリを確保します。AllocMemの引数には、配列の一つの要素の大きさに必要な要素数を掛けたものを渡します。

```
var
  i: Integer;
  IArray: PIntArray;
begin
  IArray := AllocMem(SizeOf(Integer) * 10);
```

ポインタを使って、配列の要素にアクセスします。ポインタの直後には^（ポインタ参照）が必要になるので注意してください。

```
  for i := 0 to 9 do
    IArray^[i] := i;
```

領域を使い終わったらメモリを解放します。メモリを解放するときにも、大きさが必要になるので注意してください。

```
  FreeMem(IArray, SizeOf(Integer) * 10);
```

Delphi 3では、より汎用的な方法としてバリアント型を使えます。バリアント型には任意次元で任意要素を持つ配列を定義できます。バリアント配列を定義するためには、VarArrayCreateという関数を使います。また、バリアント変数が消滅する時点で自動的に配列の内容も消去されるため、明示的にメモリを解放する必要もありません。

前述のプログラムは、バリアント配列を使えば次のように書換えられます。

```
var
  i: Integer;
  IArray: Variant;
begin
  IArray := VarArrayCreate([0, 9], varInteger);
  for i := 0 to 9 do
    IArray[I] := i;
  ...
end;
```

バリアント配列の使い方については、オンラインヘルプの「バリアント/配列と～/バリアント配列」を参照してください。

CHAP7¥DYNARRY.DPR

Q. C++の多重継承に相当するものはありますか。

Object Pascalには、多重継承はありません。C++では、多重継承によって複数のクラスが持つ機能を部品として一つの派生クラスでまとめて利用できますが、Object Pascalでは、このような機能はありません。この他、C++の（関数、演算子）オーバーロード、テンプレートなどに相当する機能がありません。

その代わりに、Object Pascalの構文はよりシンプルで覚えやすいものになっていると言えます。

Q. Borland C++やVisual C++で開発した資産を利用したいのですが、ライブラリをリンクするにはどうすればよいのでしょうか。

異なる実行ファイルやDLLの間で、互いの機能を利用する汎用的な方法としてOLEオートメーションがあります。オートメーションはOLE（COM）のインターフェースを使った手法で、きちんと対応していれば、どの開発ツールを使ったかに関わらず正しく動作します（ただし、OLEモジュールのバージョン違いによって問題が起きることもあります）。

Delphi 3ではタイプライブラリを使ったオートメーションサーバー・コントローラの作成など、オートメーションへの対応を容易にする仕組みが用意されています。これは、仮想テーブルインターフェースを使うもので、コンパイル時に型チェックできるため高速に制御できます。ただし、タイプライブラリを持たないサーバーを制御する場合は、Delphi 2.0と同じバリアント型変数を使った制御が必要です。また、Delphi 3 Standardにはタイプライブラリエディタがないため、オートメーションサーバーを作成するにはLib¥Delphi2のOleAutoユニットを使って、TAutoObjectからオートメーションクラスを継承します。

しかし、高速とはいえオートメーションでは、互いの関数のオーバーヘッドのために処理速度が犠牲になることがあります。より高速に関数を呼び出すためには、DLLによるダイナミックリンクや.OBJファイルのスタティックリンクという方法があります。DelphiからC/C++で開発したモジュールを利用するときに、.OBJファイルのスタティックリンクは制約が多いため、可能なかぎりDLLを使うことが望ましいでしょう。

Object Pascalには、Turbo Pascalからの引き継がれているコンパイル指令として{$L filename}というコンパイル指令があり、.OBJファイルをスタティックリンクできます。

ごく単純な例を挙げます。まず、次のプログラムをBorland C++ 5.0のコマンドラインコンパイラ（BCC32）で-cオプションのみでコンパイルします（-Wオプションは指定しません）。

```
int __stdcall Multiply(int a, int b) { return a * b; }
```

次に、Object Pascalで次のように記述するとCで整数乗算をする単純な関数をObject Pascalで利用できるようになります。

```
{$L MULTIPLY.OBJ}
function Multiply(a, b: Integer): Integer; stdcall; external;
```

しかし、資産と呼ばれるようなより複雑で大きなモジュールをC/C++で開発している場合は、ほとんどの場合でヘルパルーチンといった独自の内部ルーチンを呼び出します。

また、クラスライブラリやランタイムライブラリを使ったものであれば、それらのライブラリモジュールも必要となります。

DLLを使えば、こうした問題はありません。DLLは、モジュールが必要とするヘルパルーチンやライブラリが組み込まれる（または別のDLLへの参照が登録される）ため、他の開発ツールが共通のヘルパルーチンやライブラリ関数を持っている必要はありません。また、DelphiやBorland C++/Visual C++は、それぞれエクスポート関数を持つDLLを開発できるため、お互いに作成したプログラムを利用できます。

また、Delphi 3とBorland C++に限れば、両者の間での呼び出し形式やクラスオブジェクト管理における共通性は高く、相互に利用しやすくなっています。たとえば、Delphi 3とBorland C++で使われる呼び出し形式は、次のように対応しています。

Delphi 3	Borland C++ 5.0	説明
stdcall	**__stdcall**	Win32の一般的な呼び出し形式
(デフォルト)	**__fastcall**	レジスタ渡しによる高速な呼び出し形式
cdecl	(デフォルト)	C/C++の古典的な呼び出し形式
pascal	**__pascal**	Pascalの古典的な呼び出し形式

これらの呼び出し形式と前述の型の対応に注意すれば、C/C++のモジュールとDelphiのモジュールは相互に関数を利用できます。ただし、C++プログラムでは関数にextern "C"を付けて、C形式の外部名を使うようにします。C++形式では、Multiply$qqsiiのようにPascal識別子として不正な外部名になってしまいます。また、.OBJファイルをスタティックにリンクする場合、原則としてライブラリ関数を使えません。

クラスオブジェクトを利用する場合はDLLを使うべきです。オブジェクトは、クラスを定義する側で生成・解放するサポートルーチンを定義しなければなりません。たとえば、C/C++でオブジェクトを生成・解放するためのnew/deleteは、C++のライブラリにしか含まれていないため、スタティックリンクを使おうとすると、これらの関数が未定義になってしまいます。

また、C++ではクラスを多重継承したり同名の関数をオーバーロードできますが、Object Pascalではこうしたことはできません。共通で使いたいクラスでは、こうしたC++特有の機能を使わないようにします。

以下に、C++側でクラスを定義する例を示します。オブジェクトの生成と解放のためのサポート関数も合わせて宣言しています。呼び出し形式には、Win32で一般的に使われる**stdcall**を使っています。

```
class TPerson {
    char FName[16];
    int FAge;
public:
    TPerson(char* AName, int AnAge);
    virtual void __stdcall Display(char* ATitle);
    virtual char* __stdcall GetName(void);
```

```
    virtual int __stdcall GetAge(void);
};

extern "C" {
TPerson* __stdcall __export CreatePerson(char *AName, int AnAge)
    { return new TPerson(AName, AnAge); }
void __stdcall __export FreePerson(TPerson* APerson)
    { delete APerson; }
};
```

対応するObject Pascalのプログラムは、次のようになります。フィールドやメソッドの定義が、C++でのクラス定義にそのまま対応していることに注意してください。また、仮想宣言（**virtual**）や呼び出し形式（**stdcall**）を合わせている他、**abstract**で抽象メソッドであることを宣言しています。これは、このメソッドの定義が、Object Pascal側では定義されないためです。オブジェクトの生成と解放のためのルーチンは、DLLからインポートしています。

```
type
  TPerson = class
    FName: array [0..15] of Char;
    FAge: Integer;
    procedure Display(ATitle: PChar); virtual; stdcall; abstract;
    function GetName: PChar; virtual; stdcall; abstract;
    function GetAge: Integer; virtual; stdcall; abstract;
  end;

function CreatePerson(AName: PChar; AnAge: Integer): TPerson; stdcall;
         external 'CPPCLASS.DLL';
procedure FreePerson(APerson: TPerson); stdcall;
         external 'CPPCLASS.DLL';
```

これらを組み合わせることで、C++側のクラスオブジェクトをObject Pascal側で利用できます。より複雑なクラスで両者を組み合わせる場合は、高度なテクニックが要求されます。

CHAP7¥USECMOD.DPR
CHAP7¥MAKEDLL.BAT

Q. Delphiで作成したフォームやクラスをC/C++などの他の処理系で利用できますか。

前項目と同じくOLEオートメーションを使うことで、Delphiで作成したアプリケーションとC/C++で作成したアプリケーションで、お互いの機能を利用できます。オートメーションを使わない場合は、.OBJのスタティックリンクやDLLによるダイナミックリンクを使うことになります。

Delphiでは、デフォルトで.DCUという拡張子のユニットオブジェクトを生成します。これは、Delphi特有のファイルでC/C++からは利用できません。しかし、**[プロジェクト(P)｜オプション(O)｜リンカ]**で**[.OBJファイルを作成(O)]**を選ぶと、.OBJファイルを生成できるようになります。ただし、Delphiのフォームやクラスを利用する場合には、生成された.OBJファイルをスタティックリンクしようとしても、リンクエラーになります。フォームやクラスを利用する場合には、DLLを使います。

以下に、Delphiでクラスを定義する例を示します。この場合は、Delphiのプログラムでオブジェクトを生成するルーチンを定義します。

```
type
  TPerson = class
    FName: array [0..15] of Char;
    FAge: Integer;
    constructor Create(AName: PChar; AnAge: Integer);
    procedure Display(ATitle: PChar); virtual; stdcall;
    function GetName: PChar; virtual; stdcall;
    function GetAge: Integer; virtual; stdcall;
  end;

function CreatePerson(AName: PChar; AnAge: Integer)
         : TPerson; stdcall;
procedure FreePerson(APerson: TPerson); stdcall;
```

対応するC++のプログラムは次のようになります。Object Pascalメソッドに対応するC++のメンバ関数は、純粋仮想関数（= 0）として定義されています。

```
class TPerson {
    char FName[16];
    int FAge;
public:
    virtual void __stdcall Display(char* ATitle) = 0;
    virtual char* __stdcall GetName(void) = 0;
    virtual int __stdcall GetAge(void) = 0;
};

extern "C" {
TPerson* __stdcall CreatePerson(char *AName, int AnAge);
```

```
void __stdcall FreePerson(TPerson* APerson);
};
```

CHAP7¥RICHED.DPR
CHAP7¥MAKEEXE.BAT

付録

付録CD-ROMについて
用語解説

付録CD-ROMについて

付録ディスクに含まれるプログラムは、「Delphi 3 Q & A 150選」の理解を助けるためのプログラムです。ファイルは、QABOOK3フォルダ（ディレクトリ）の中にあり、すべてのファイルはFILELIST.TXTに記載されています。

付録ディスクに含まれるプログラムやドキュメントの一部または全部を無断で転載、引用することを禁じます。ただし、プログラム例をDelphiで作成するアプリケーションに組み込むことは自由です。なお、付録ディスクに含まれるプログラムによるいかなる影響についても著者、株式会社ビレッジセンター、ボーランド株式会社はその責を負いません。

オリジナルコンポーネント

本書のサンプルプロジェクトのうち、いくつかのプロジェクトではQACOMPOディレクトリに含まれるコンポーネントが必要です。これらのプロジェクトを試す前にQACOMPOディレクトリにあるコンポーネントを登録しておくとよいでしょう。すべてのコンポーネントを登録するためには、Delphiの[コンポーネント(C)｜パッケージのインストール(P)]で表示されるダイアログボックスで[追加(A)]ボタンを押し、¥QABOOK3¥QACOMPOディレクトリのdclqa30.dplを追加します。

サンプルプログラムの内容

すべてのプロジェクトファイル、ヘルプファイル、コンポーネントユニットは以下のとおりです（フォルダQABOOK3の記述は省略してあります）。

ファイル名	ページ	内容
(第1章　統合開発環境)		
CHAP1¥CPUVIEW.DPR	27	CPUウィンドウの設定
CHAP1¥SMALLAPP.DPR	25	小さいアプリケーション
CHAP1¥USEDMSG.DPR	32	デバッグ用コンポーネントの使用例
(第2章　アプリケーション／フォーム)		
CHAP2¥ANIMFRM.DPR	64	フォームを最小化する時のアニメーション動作
CHAP2¥APPICON.DPR	52	アプリケーションのアイコン
CHAP2¥CHGCSR.DPR	50	カーソルの変更
CHAP2¥DISPARG.DPR	37	コマンドライン引数の表示
CHAP2¥DRGFILE.DPR	61	エクスプローラからのドラッグ＆ドロップ
CHAP2¥DYNFORM.DPR	38	選択可能なフォームを使う
CHAP2¥ELLIPFRM.DPR	54	長方形でないフォーム
CHAP2¥ENDSESS.DPR	55	Windowsの終了を検出する
CHAP2¥EXEPATH.DPR	36	起動ディレクトリパスの表示
CHAP2¥FIXPOS.DPR	56	移動できないフォーム
CHAP2¥FMSCALE.DPR	46	フォームのスケール
CHAP2¥FRMPROJ.DPR	34	複数のフォームを使うプロジェクト
CHAP2¥KEEPMIN.DPR	62	フォームのアイコン状態を維持
CHAP2¥MDIDRAW.DPR	58	MDIの背景に描画する
CHAP2¥MDIDRAW.DPR	59	MDIのスクロールバーを表示しない
CHAP2¥MDIDRAW.DPR	59	MDIのウィンドウメニューを変更する
CHAP2¥MDIHELP.DPR	60	MDIとヘルプの呼び出し
CHAP2¥MDIPROG.DPR	44	ShowModalとフォームメモリの解放
CHAP2¥NOTITLE.DPR	52	タイトルバーのないフォーム
CHAP2¥NOTITLE.DPR	53	クライアント領域を使ったフォームの移動
CHAP2¥ONLYONE.DPR	35	二重起動の禁止
CHAP2¥ORIGCSR.DPR	51	独自カーソルの使用
CHAP2¥PROGRESS.DPR	40	オープニングダイアログ（スプラッシュ画面）
CHAP2¥RESIZEW.DPR	57	サイズ変更が限定されたフォーム
CHAP2¥SCRLPOS.DPR	47	フォームのスクロール位置
CHAP2¥SMENU.DPR	63	システムメニューに項目を追加
CHAP2¥SPLASH.DPR	40	オープニングダイアログ（スプラッシュ画面）
CHAP2¥USEPAPP.DPR	45	PseudoAppの使用例
CHAP2¥USEPAPP.DPR	45	バルーンヒント表示
CHAP2¥USETRAY.DPR	63	タスクトレイにアイコンを登録
CHAP2¥USEWPLC.DPR	48	レジストリを使ったフォーム位置の保存と復帰
CHAP2¥USEWPLC.DPR	49	レジストリを使ったフォント情報の保存と復帰

ファイル名	ページ	内容
(第3章　プログラミング)		
CHAP3¥ANGLESTR.DPR	74	文字列を斜めに描画
CHAP3¥BLUEFRM.DPR	70	ビットマップの代入
CHAP3¥CHGPROP.DPR	67	プロパティの変更
CHAP3¥CHKCHAR.DPR	86	文字判定ルーチン
CHAP3¥CONAPP.DPR	84	コンソールアプリケーション
CHAP3¥CVMSBIN.DPR	94	MSBIN形式の変換
CHAP3¥DAYNAMES.DPR	69	月名・曜日名で英単語を使用
CHAP3¥DESKBMP.DPR	70	スクリーン全体のビットマップを取得
CHAP3¥DRAWIMG.DPR	73	ビットマップの一部を透明にする
CHAP3¥ENUMWIN.DPR	87	コールバック関数
CHAP3¥EXECCMD.DPR	77	実行ファイルの呼び出し
CHAP3¥EXITWIN.DPR	79	Windowsの再起動
CHAP3¥FILER.DPR	91	ファイルをごみ箱へ移動する
CHAP3¥IDENTFIL.DPR	88	ファイルアクセス
CHAP3¥IFELSE.DPR	66	セミコロンの付け方
CHAP3¥IOPORT.DPR	85	I/Oポートへの入出力
CHAP3¥MKMETAS.DPR	72	メタファイルの作成
CHAP3¥PRTEST.DPR	80	印字方向の指定
CHAP3¥PRTEST.DPR	80	印字フォントの指定
CHAP3¥PRTEST.DPR	80	用紙トレイの設定
CHAP3¥PRTEST.DPR	82	イメージの印刷
CHAP3¥PRTEST.DPR	83	印刷プレビューの表示
CHAP3¥PRTEXT.DPR	84	テキストの印刷
CHAP3¥RBUTTONS.DPR	68	複数のラジオボタンのグループ
CHAP3¥SCONST.DPR	67	文字列定数とシングルクォート
CHAP3¥SETFORE.DPR	68	アクティブなウィンドウの設定
CHAP3¥VIEWINI.DPR	88	ファイルアクセス
CHAP3¥WEEKWALL.DPR	76	曜日ごとに壁紙を変更する
CHAP3¥WEEKWALL.DPR	92	スタートアップショートカットを作成
CHAP3¥WFOCUS.DPR	68	重複する識別子の参照方法
CHAP3¥WINDIR.DPR	95	バッファとしての文字列型
(第4章　コンポーネント)		
CHAP4¥ACTPAGE.DPR	126	PageControlのページ表示
CHAP4¥APPEXCEP.DPR	124	コンポーネントが発生する例外
CHAP4¥BIGEDIT.DPR	112	大規模テキストの編集
CHAP4¥BIGEDIT.DPR	113	テキストの編集とカーソル位置の移動
CHAP4¥BIGEDIT.DPR	113	指定行への移動

ファイル名	ページ	内容
CHAP4¥BIGEDIT.DPR	114	テキストの編集と上書きモード
CHAP4¥BIGEDIT.DPR	115	テキストの編集と文字列検索
CHAP4¥CALCDEMO.DPR	116	１行入力における右寄せ表示
CHAP4¥DBLCLK.DPR	106	クリックとダブルクリックの区別
CHAP4¥DRAWIMG.DPR	122	Imageコンポーネントへの描画
CHAP4¥DYNEDIT.DPR	120	コンポーネントの動的作成
CHAP4¥EDITMOV.DPR	107	矢印キーでコンポーネントを移動する
CHAP4¥ODLBOX.DPR	110	リストボックスの選択項目の色指定
CHAP4¥RTEDIT.DPR	115	RichEditでの上付き指定、下付き指定
CHAP4¥RTEDIT.DPR	117	RichEditでの縦書き編集
CHAP4¥SGRIDML.DPR	104	文字列グリッドと複数行の表示
CHAP4¥STOPWATC.DPR	123	マルチメディアタイマーを使ったストップウォッチ
CHAP4¥STRGRID.DPR	101	文字列グリッドの選択セルの色指定
CHAP4¥STRGRID.DPR	103	文字列グリッドのセル単位での色指定
CHAP4¥STRGRID.DPR	105	文字列グリッドと固定セルのクリック
CHAP4¥TABPAGE.DPR	126	タブの左右への割り当て
CHAP4¥TABPAGE.DPR	127	タブのオーナー描画
CHAP4¥USEBTNEX.DPR	98	複数行のキャプションや色を使ったボタン
CHAP4¥USECOMPO.DPR	116	ヨミガナの取りだし
CHAP4¥USESBAR.DPR	99	スクロールバーのつまみの幅を変える
CHAP4¥USETEXT.DPR	118	StaticTextの使用例
CHAP4¥ZOOMPNL.DPR	100	コンポーネントの描画
CHAP4¥ZORDER.DPR	119	実行時のＺオーダーの変更
CHAP4¥ZORDER.DPR	119	実行時のＺオーダーの確認
(第5章　データベース)		
CHAP5¥CHKDBEX.DPR	152	Paradoxテーブルのリモート更新チェック
CHAP5¥CHKDBEX.DPR	152	問い合わせの進行状況を表示
CHAP5¥CHKDBG.DPR	135	DBGridにおけるスクロールバーの消去
CHAP5¥CONVTXT.DPR	143	テーブル形式の変換
CHAP5¥CSVTODB.DPR	144	テキスト形式のデータを変換
CHAP5¥DBALIAS.DPR	148	ユーザー定義のデータベースエリアス
CHAP5¥DBGMULT.DPR	132	DBGridにおける複数のテーブル表示
CHAP5¥DBGMREC.DPR	134	DBGridにおける複数レコードの選択
CHAP5¥DRAWCELL.DPR	131	DBGridにおける選択中のセルの描画
CHAP5¥DYNTABLE.DPR	136	テーブルコンポーネントを実行時に作成する
CHAP5¥FINDREC.DPR	148	検索の高速化
CHAP5¥MKTABLE.DPR	138	新しいテーブルの作成
CHAP5¥MKTABLE.DPR	141	テーブルにインデックスを付ける

ファイル名	ページ	内容
CHAP5¥NAVDSET.DPR	149	DataSourceを使ったレコード操作
CHAP5¥PACKDB.DPR	150	dBASE/Paradoxテーブルの圧縮
CHAP5¥PACKDB.DPR	151	暗号化されたdBASEテーブルのオープン
CHAP5¥TBLRANGE.DPR	146	テーブルの範囲指定
CHAP5¥USENAVB.DPR	150	レコード操作用の個別ボタン
(第6章　for Visual Basicプログラマ)		
CHAP6¥AREDRAW.DPR	162	Visual BasicのAutoRedrawプロパティの代用
CHAP6¥CTLARRY.DPR	158	Visual Basicのコントロール配列の代用
CHAP6¥FORSTEP.DPR	166	Visual BasicのFor～Stepの代用
CHAP6¥MKVBDAT.VBP	169	Visual Basicのデータの利用
CHAP6¥PASSVAL.VBP	179	Visual Basicプログラムとの連携
CHAP6¥PROCMSG.DPR	157	Visual BasicのDoEventsの代用
CHAP6¥RDVBDAT.DPR	169	Visual Basicのデータの利用
CHAP6¥USEBEEP.DPR	164	Visual Basicのジェネラルプロシージャの代用
CHAP6¥USELINE.DPR	165	ライン（直線）コントロール
CHAP6¥USESCALE.DPR	167	Visual Basicのスケール機能の代用
CHAP6¥VBVALUE.DPR	179	Visual Basicプログラムとの連携
(第7章　for C/C++プログラマ)		
CHAP7¥DISPFMT.DPR	193	書式付き文字列処理
CHAP7¥DYNARRY.DPR	197	動的配列の確保
CHAP7¥FACTOR.DPR	184	C/C++のreturn文の代替
CHAP7¥MAKEDLL.BAT	199	C/C++資産の活用
CHAP7¥MAKEEXE.BAT	202	C/C++からDelphiフォームの呼び出し
CHAP7¥METHODP.DPR	189	メンバへのポインタ
CHAP7¥OPENARR.DPR	194	可変個引数を使った手続き
CHAP7¥RICHED.DPR	202	C/C++からDelphiフォームの呼び出し
CHAP7¥USECMOD.DPR	199	C/C++資産の活用
(オリジナルコンポーネント)		
QACOMPO¥CTRLSEX.PAS		既存のコンポーネントの拡張
QACOMPO¥DBEXCTLS.PAS		データベース対応コンポーネント
QACOMPO¥DEBUGMSG.PAS		デバッグ用コンポーネント
QACOMPO¥NEWCTRLS.PAS		新しいコンポーネント
QACOMPO¥SYSCTRLS.PAS		システム関係のコンポーネント

用語解説

DLL

実行時に共有できるライブラリファイル（.DLL）。ダイナミックリンクライブラリ（Dynamic Link Library）。Delphiでは、プロジェクトソースの先頭をprogramではなくlibraryにすることで作成できる。コンパイル・リンクして作成するという点で実行ファイル（.EXE）に似ているが、単独では使えない。必ず他の実行ファイルやDLLとともに使う。

Win32

Windows95やWindows NTのような32ビットのアプリケーションを実行させるためのWindows環境。

Windows API

Windows自身が提供する機能を使うための関数。APIは、Application Program Interfaceの略。

イベント

コンポーネント（またはフォーム）に対する動作。マウスやキーボードからの入力、表示・消去などのきっかけによって発生する。イベントを処理するプログラムのことをイベントハンドラと呼ぶ。

エクスポート関数

他の実行ファイルやDLLから呼び出される関数。

オブジェクト

クラスをもとに生成された実体。クラスを設計図とすれば、オブジェクトは設計図を元に作られた物体である。一般に、ひとつのクラスから複数のオブジェクトを生成できる。Delphiでは、TButtonなどのコンポーネントはクラスであり、フォーム上に配置されたButton1、Button2などがオブジェクトとなる。

オブジェクトインスペクタ

フォームやフォーム上に配置したコンポーネントのプロパティやイベントハンドラを定義するためのウィンドウ。通常、画面の左側に表示されている。オブジェクトインスペクタの上部にあるコンボボックスは、オブジェクトセレクタと呼び、フォーム上のすべてのコンポーネントから目的のものを選択できる。

関数

ひとまとまりの処理や計算を記述し、他から呼び出して値を返すように定義されたもの。Object Pascalでは、functionという予約語で定義する。

クラス

ある目的のためのデータや手続き・関数をまとめて定義したもの。クラスを利用するためには、オブジェクトを生成しなければならない。Delphiでは、フォームやコンポーネントはすべてクラスとして定義されている。

コントロール

コンポーネントと同義。特に実行時に目に見える（ビジュアル）コンポーネントのことを指す。

コンパイル

人間が理解できるプログラミング言語をコンピュータが理解できる機械語に変換すること。Delphiでは、Object Pascalで記述されたプロジェクトソース（.DPR）またはユニット（.PAS）を解析して、ユニットバイナリ（.DCU）や実行ファイル（.EXE）を作成すること。

コンポーネント

フォーム上に配置して、ユーザーインターフェースを作成するための部品。Delphiの開発環境では、コンポーネントパレットに登録されている。コンポーネントは非ビジュアルであるものとビジュアルであるものに大別される。非ビジュアルコンポーネントは、設計時にはプロパティを設定するために表示されているが実行時には目に見えない。ビジュアルコンポーネントは、設計時にも実行時にも表示される。

スピードメニュー

マウスの右ボタンをクリックして表示されるメニュー。ポップアップメニュー。

スマートリンク

リンクするときにプログラムの中で使われていないルーチンを実行ファイル（.EXE）に含めないこと。実行ファイルの縮小化のために役立つ。

ディレクトリ

ディスク上でファイルやプログラムが保存されている場所。フォルダと同義。

テキストファイル

一般に、人が読んで理解できる形式のファイル。英数字、かな、漢字、記号などで書かれたファイル。メモ帳などで表示できる。

手続き

ひとまとまりの処理を記述し、他から呼び出せるようにしたもの。Object Pascalでは、procedureという予約語で定義する。

デフォルト

特に変更していない状態。

バイナリファイル

コンピュータが理解する形式のファイル。一般に、バイナリファイルはそれを利用するために専用のツールが必要となる。実行形式ファイル（.EXE）、ユニットバイナリ（.DCU）、ビットマップファイル（.BMP）などもバイナリファイルである。

フィールド

クラスのために定義された変数のこと。
これとは別に、テーブルの項目のこと。

フォーム

アプリケーションのユーザーインターフェースを設計する土台となる場所。ウィンドウのこと。
Delphiでは、ひとつのフォームに2種類のファイルが対応する。
ひとつは、バイナリイメージを保持する.DFMファイルで、フォームのビジュアルなイメージを保持する。もうひとつは、ソースコードを保持する.PASファイル（フォームユニット）であり、イベントハンドラなどのプログラムを記述する。フォームユニットは、コードエディタで編集する。

フォルダ

ディスク上でファイルやプログラムが保存されている場所。ディレクトリと同義。

プロジェクト

アプリケーションを作成するために必要なファイルの集まり。プロジェクトに関係するファイルを管理するファイルをプロジェクトファイルと呼ぶ。Delphiでは、プロジェクトファイルは拡張子が.DPRというObject Pascalのプログラムになっており、[表示(V)｜ﾌﾟﾛｼﾞｪｸﾄｿｰｽ(J)]で表示できる。

プロジェクトソース

プロジェクトに関わるファイルを管理するプログラム。拡張子が.DPRのファイル。

プロパティ

コンポーネント（またはフォーム）の性質を決める情報のこと。フォームの色、ラベルに表示するテキスト、ボタンの種類などがプロパティとして用意されている。

メソッド

クラスのために定義されたルーチンのこと。

リソース

ビットマップなどリンク時に実行ファイル（.EXE）に組み込まれるデータまたはデータファイル（.RES）のこと。ビットマップ、カーソル、アイコンなどのイメージリソースはImage Editorで作成できる。Delphiのプログラムでリソースファイルを取り込むためには、{$R filename}または{$RESOURCE filename}とする。
これとは別に、メモリやハードディスクなどコンピュータが提供する資源のこと。

リンク

複数のユニットオブジェクト（.DCU）から実行ファイル（.EXE）を作成すること。Delphiでは、プロジェクトソースのコンパイルとリンクが同時に行なわれる。

ルーチン

ひとまとまりの処理を記述し、他から呼び出せるようにしたもの。関数または手続き。

ユニット

プログラムを記述するソースファイルの単位。拡張子は.PAS。unitという予約語ではじまり、外部から参照する際の仕様を記述するinterface部、ユニット内部で使う変数の定義やルーチンの本体を定義するimplementation部、ユニットを初期化するときのプログラムを定義するinitialization部、ユニットが終了するときのプログラムを定義するfinalization部から構成される。
あるいは、ユニットをコンパイルして生成されたバイナリファイル（.DCU）のこと。

コンポーネント・リファレンス

TAnimateForm

ユニット: SysCtrls

TObject ・ TPersistent ・ TComponent ・ TAnimateForm

TMDIBackは、メインフォームの最小化・復元の際のアニメーション動作を有効にします。

説明 Delphiのアプリケーションでは、アプリケーションのための隠れたウィンドウが作られ、これが内部でのメインウィンドウとなっているため、Delphiとしてのメインウィンドウであるメインフォームが最小化・復元されるときにはアニメーションが禁止されます。

TAnimateFormは、メインフォームのアニメーション動作を有効にするためのコンポーネントです。TAnimateFormをプロジェクトのメインフォームに配置すれば、最小化または最大化する際にアニメーション動作が働くようになります。

TAnimateFormは、メインフォームに配置するだけで効果を発揮するため、プロパティやメソッドはありません。

CHAP2¥ANIMFRM.DPR

TButtonEx

ユニット: CtrlsEx

TObject ・ TPersistent ・ TComponent ・ TControl ・ TWinControl ・ TButtonControl ・ TButton ・ TButtonEx

オーナー描画をサポートしたボタンです。

説明 TButtonExは、プログラムでボタン表面を描画できるオーナー描画をサポートしています。独自のスタイルのボタンを使えます。

CHAP4¥USEBTNEX.DPR

プロパティ

```
property Canvas: TCanvas;
```

説明　オーナー描画ボタンで、ボタン表面を描画する際に使うキャンバスです。

```
property ModalResult: TModalResult;
```

説明　TButtonのModalResultと同等ですが、mrNone以外に設定すると、自動的にCaptionプロパティが適切な文字列に変更されます。

```
type
    TButtonExStyle = (bsStandard, bsOwnerDraw, bsOwnerDrawWithFrame);
property Style: TButtonExStyle;
```

説明　ボタンが標準ボタンであるかオーナー描画ボタンであるかを指定します。bsOwnerDrawWithFrameを選ぶと、ボタンの枠のみ自動的に描画されるため、bsOwnerDrawよりも処理が簡単になります。

イベント

```
type
    TDrawButtonEvent = procedure (Control: TWinControl; Rect: TRect;
                       State: TOwnerDrawState) of object;
property OnDrawButton: TDrawButtonEvent;
```

説明　オーナー描画ボタンで、ボタン表面を描画するために発生するイベントです。StyleがbsOwnerDrawWithFrameのときは、ボタンの枠のみが描画され、Rectはやや小さめになります。

TCanvasScale

ユニット: NewCtrls

TObject ・ TPersistent ・ TComponent ・ TCanvasScale

TCanvasScaleは、独自の座標系を使ってCanvas描画できる仕組みを提供します。

説明　TCanvasScaleは、Canvasプロパティに独自の座標系を割り当てます。新しい座標系は、ScaleLeft、ScaleTop、ScaleWidth、ScaleHeightで指定しますが、四隅の座標で指定する場合はScaleWidth、ScaleHeightの代わりにScaleRight、ScaleBottomを使います。

TCanvasScaleは、座標系を割り当てたいCanvasを[]で囲んで使います。たとえば、フォームのキャンバスに座標系を割り当てる場合は、CanvasScale1[Canvas]とします。これで得られるキャンバスには、実数を利用できるメソッドが用意されています。

CHAP6¥USESCALE.DPR

プロパティ

```
property Def[C: TCanvas]: TScaledCanvas;
```

説明　変換された座標系を使うための新しいキャンバスオブジェクトを返します。これはデフォルトプロパティなので、実際にはCanvasScale1.Def[Canvas]とする代わりに、CanvasScale1[Canvas]と記述できます。

```
property ScaleBottom: Double;
```

説明　新しい座標系の下端を指定します。

```
property ScaleHeight: Double;
```

説明　新しい座標系の高さを指定します。

```
property ScaleLeft: Double;
```

説明　新しい座標系の左端の値を指定します。

```
property ScaleRight: Double;
```

説明　新しい座標系の右端を指定します。

```
property ScaleTop: Double;
```

説明　新しい座標系の上端の値を指定します。

```
property ScaleWidth: Double;
```

説明　新しい座標系の幅を指定します。

メソッド

```
procedure SetScaleBounds(ALeft, ATop, ARight, ABottom: Double);
```

説明　新しい座標系を左上の座標と幅・高さで指定します。

```
procedure SetScaleRect(ALeft, ATop, ARight, ABottom: Double);
```

説明　新しい座標系を四隅の座標で指定します。

TScaledCanvasのメソッド

```
type
    TPointReal = record
      X, Y: Double;
    end;

    TRectReal = record
    case Integer of
      0: (Left, Top, Right, Bottom: Double);
      1: (TopLeft, BottomRight: TPointReal);
    end;

procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Double);
procedure BrushCopy(const Dest: TRectReal; Bitmap: TBitmap;
          const Source: TRectReal; Color: TColor);
procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Double);
procedure Draw(X, Y: Double; Graphic: TGraphic);
procedure DrawFocusRect(const Rect: TRectReal);
procedure Ellipse(X1, Y1, X2, Y2: Double);
procedure FillRect(const Rect: TRectReal);
procedure FloodFill(X, Y: Double; Color: TColor;
          FillStyle: TFillStyle);
procedure FrameRect(const Rect: TRectReal);
procedure MoveTo(x, y: Double);
procedure LineTo(x, y: Double);
procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Double);
procedure Rectangle(X1, Y1, X2, Y2: Double);
procedure Refresh;
```

```
procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Double);
procedure StretchDraw(const Rect: TRectReal; Graphic: TGraphic);
function TextHeight(const Text: string): Double;
procedure TextOut(X, Y: Double; const Text: string);
procedure TextRect(Rect: TRectReal; X, Y: Integer; const Text: string);
function TextWidth(const Text: string): Double;
function Brush: TBrush;
function Font: TFont;
function Pen: TPen;
```

説明　TScaledCanvasは、TCanvasScaleコンポーネントのDefプロパティが返すオブジェクト型です。これらのメソッドは、TCanvasの同名のメソッド（またはプロパティ）とほぼ同等の機能を持ちます。ただし、座標系には実数型（Double）が使われる点に注意してください。

```
function GetCopyMode: TCopyMode;
procedure SetCopyMode(Value: TCopyMode);
```

説明　TCanvasのCopyModeプロパティの代わりに使います。

```
function GetPixel(X, Y: Double): TColor;
procedure SetPixel(X, Y: Double; C: TColor);
```

説明　TCanvasのPixelsプロパティの代わりに使います。

```
procedure GetPenPos: TPointReal;
```

説明　TCanvasのPenPosプロパティの代わりに使います。PenPosを設定するためには、MoveToメソッドを使います。

関連する手続き・関数

```
function BoundsReal(ALeft, ATop, AWidth, AHeight: Double): TRectReal;
```

説明　与えられた引数から、長方形の領域をあらわすTRectReal型のレコード値を返します。第3、4引数は右下の座標値ではなく、幅と高さをとります。

```
function PointReal(AX, AY: Double): TPointReal;
```

説明　与えられた引数から、座標をあらわすTPointReal型のレコード値を返します。

```
function RectReal(ALeft, ATop, ARight, ABottom: Double): TRectReal;
```

説明　与えられた引数から、長方形の領域をあらわすTRectReal型のレコード値を返します。

TCompoString

ユニット: NewCtrls

TObject・TPersistent・TComponent・TCompoString

TCompoStringは、IMEを使って入力されたときのヨミガナを自動的に取得するコンポーネントです。

説明　TCompoStringは、Controlプロパティに指定されたコントロールにIME（かな漢字変換機能）を使って文字を入力したとき、OnCompositionStrイベントを発生させて使われた構成文字列（ヨミガナ）を渡します。

CHAP4¥USECOMPO.DPR

プロパティ

```
property Control: TControl;
```

説明　ヨミガナを取り出したいコントロールを指定します。指定したコントロールにIMEを使って文字列が入力されるとOnCompositionStrイベントが発生します。

イベント

```
type
    procedure TCompositionStrEvent = procedure (Sender: TObject;
              Value: string) of object;
    property OnCompositionStr: TCompositionStrEvent;
```

説明　Controlプロパティに指定されたコントロールにIMEを使って文字列が入力されるときに発生するイベントです。IMEに渡された構成文字列がValue引数に渡されます。

TCoolHint

ユニット: SysCtrls

TObject・TPersistent・TComponent・TCoolHint

TCoolHintは、長方形以外の方法でヒントを表示させたい場合に使うコンポーネントです。

説明　TCoolHintは、長方形以外の方法でヒントを表示させたい場合に使うコンポーネントです。風船のような外観のヒント表示（htBalloon）と角の丸い長方形（htRoundRect）を設定できる他、イベントハンドラを処理して独自のスタイルでヒントを表示できます。

CHAP2¥USEPAPP.DPR

プロパティ

```
property Active: Boolean;
```

説明　特殊なヒント表示を有効にするかどうか指定します。

```
type
    THintType = (htOwnerDraw, htOwnerDrawTransparent, htBalloon,
                 htRoundRect);
property HintType: THintType;
```

説明　ヒント表示のスタイルを設定します。

値	意味
htOwnerDraw	OnActivateとOnPaintを処理して、プログラムでヒントを加工、描画します。ヒントの背景（長方形）のみ描画されます。
htOwnerDrawTransparent	OnActivateとOnPaintを処理して、プログラムでヒントを加工、描画します。ヒントの背景も描画されないため、自由な形式でヒントを表示できます。
htBalloon	バルーン形式でヒントを表示します。
htRoundRect	角の丸い長方形でヒントを表示します。

イベント

```
type
    THintActivateEvent = procedure (var Rect: TRect;
                         const AHint: string) of object;
    property OnActivate: THintActivateEvent;
```

説明　ヒントが有効になった時点で発生するイベントです。Rect引数には、デフォルトの大

きさが渡されますが、イベントハンドラでこの大きさを変更できます。AHintには、ヒント表示するための文字列が渡されます。

```
type
    THintPaintEvent = procedure (AHintWnd: THintWindow) of object;
    property OnPaint: THintPaintEvent;
```

説明　ヒントを描画する必要があるときに発生するイベントです。AHintWndは、ヒントウィンドウ自身が渡されます。THintWindow型に定義されている情報を利用してヒントを描画します。

TDBGridEx

ユニット: DBExCtls

TObject・TPersistent・TComponent・TControl・TWinControl・TCustomControl・TCustomGrid・TCustomDBGrid・TDBGrid・TDBGridEx

スクロールバーの表示を制御できるデータベースグリッドです。

説明　TDBGridでは、必要に応じてスクロールバーの表示が自動的に制御されます。TDBGridExでは、ScrollBarsプロパティによって縦横のスクロールバーが表示されないように設定できます。

CHAP5¥CHKDBG.DPR

プロパティ

```
property ScrollBars: TScrollStyle;
```

説明　縦横のスクロールバーを表示するかどうかを指定します。ScrollBarsの指定に関わらず、グリッドがデータを表示するためにスクロールバーが不要な場合には、スクロールバーは表示されません。

TDBNavButton

ユニット: DBExCtls

TObject・TPersistent・TComponent・TControl・TWinControl・TButtonControl・TButton・TDBNavButton

データセットのレコードの移動や挿入・削除を行なうためのボタンです。

説明　TDBNavButtonは、TDBNavigatorに似ていますが、ひとつのボタンがひとつの動作に結び付いています。ボタンの動作は、NavTypeプロパティで指定します。

CHAP5¥USENAVB.DPR

プロパティ

```
property DataSource: TDataSource;
```

説明　操作したいデータセットが割り当てられているデータソースコンポーネントを指定します。

```
type
    TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert,
                    nbDelete, nbEdit, nbPost, nbCancel, nbRefresh);
property NavType: TNavigateBtn;
```

説明　データセットをどのように操作するかを指定します。NavTypeを指定することで、自動的にボタンのキャプションが設定されます。

値	意味
nbFirst	最初のレコード
nbPrior	前のレコード
nbNext	次のレコード
nbLast	最後のレコード
nbInsert	レコードの挿入
nbDelete	レコードの削除
nbEdit	レコードの編集
nbPost	レコードの登録
nbCancel	編集の取り消し
nbRefresh	データの更新

TDebugMsg

ユニット: DebugMsg

TObject ・ TPersistent ・ TComponent ・ TDebugMsg

Assert手続きによる例外を捕捉し、デバッグ用のメッセージを表示します。

説明　DebugMsgは、Assert手続きが発生する例外（EAssertionFailed）をトラップし、独自のウィンドウに表示するためのデバッグ用コンポーネントです。

Delphi 3では、Assertというデバッグ用の手続きが用意されており、プログラム中に必ず真となるはずの条件式を与え、条件式が偽になった場合に例外を発生して通知できます。(Assert手続きの詳細はオンラインヘルプを参照してください)

DebugMsgはプロジェクト中の任意のフォームに1個だけ配置します。DebugMsgを配置すると、プログラムを実行するときにメッセージ表示用のウィンドウを表示され、EAssertionFailed例外で得られるメッセージを表示できるようになります。このとき、[ツール(T)|環境オプション(O)]の「例外でデバッガを開く(B)]のチェックを外しておいてください。この項目がチェックされていると開発環境で例外が発生するたびに、プログラムが停止します。

たとえば、Assert(PBuf <> nil, 'Transfer Buffer is nil');と記述しておくと、PBufがnilの場合に'Transfer Buffer is nil'というメッセージがウィンドウに記録されます。常にメッセージを記録させたい場合は、Assert(false, 'Check Point');などと記述します。

Assertを使う利点として、$IFDEF ～ $ENDIF指令を使わなくても、{$C-}（または{$ASSERTIONS OFF}を記述するだけで、リリース版からデバッグ用のコードを取り除けるということがあります。プロジェクト全体でAssertを無効化するには、[プロジェクト(P)｜オプション(O)]の[コンパイラ]ページで[アサートの使用(C)]チェックをオフにします。また、リリース版では、DebugMsgコンポーネントも削除してください。

Assertでは、自動的にソースファイル名と行数がメッセージに追加されるため、エラーが発生した箇所を特定しやすくなります。たとえば、Assert(false);と記述しておくだけでファイル名と行番号を記録できます。

また、次のようにメッセージ部分でFormat関数を使えば情報を書式化して表示できます。

```
Assert(false, Format('InfoNo = %d', [InfoNo]));
```

メッセージウィンドウでは、ログの保存および消去のためのボタンがあります。また、ツールバーで右クリックすれば常に手前に表示しないようにしたり、EAssertionFailed以外の例外も捕捉できるようになります。ログをカット＆ペーストは、ログウィンドウ上で右クリックしてください。メッセージウィンドウは間違えて消してしまわないように、閉じられないようになっています。

なお、DebugMsgではApplication.OnExceptionを設定しているため、

Application.OnExceptionを設定しているアプリケーションでは期待通りに動作しないことがあります。

CHAP1¥USEDMSG.DPR

プロパティ

```
property AllExceptions: Boolean;
```

説明　EAssertionFailedを含む、すべての例外を捕捉するかどうかを指定します。AllExceptionsがFalseのときは、EAssertionFailedのみを捕捉します。

```
property StayOnTop: Boolean;
```

説明　デバッグ用のメッセージウィンドウを常に前面に表示するかどうかを指定します。

TEditEx

ユニット: CtrlsEx

TObject・TPersistent・TComponent・TControl・TWinControl・TCustomEdit・TEdit・TEditEx

TEditExは、右寄せ入力やカンマ表示をサポートします。

説明　TEditExでは、右寄せや中央揃えでの入力、3桁ごとのカンマ表示をサポートした1行入力コントロールです。電卓の数値入力のように、入力する文字列を右側せしたい場合に使えます。

CHAP4¥CALCDEMO.DPR

プロパティ

```
property Alignment: TAlignment;
```

説明　編集する文字列の位置を指定します。

値	意味
taLeftJustify	左寄せ（デフォルト）
taCenter	中央揃え
taRightJustify	右寄せ

```
property EditText: string;
```

説明　Thousandsプロパティが Trueのときも、表示されている文字列そのものをあらわします。

```
property Text: string;
```

説明　Thousandsプロパティが Trueのときは、カンマを取り除いた文字列をあらわします。Thousandsプロパティが Falseのときは、EditTextと同じです。

```
property Thousands: Boolean;
```

説明　3桁ごとにカンマを挿入するかどうかを示します。Thousandsが Trueのときは、文字を入力するたびにテキストを評価して3桁ごとにカンマを挿入します。ただし、文字列が削除される場合は適切な位置に調整されないことがあります。

TFileDrop

ユニット: NewCtrls

TObject・TPersistent・TComponent・TFileDrop

TFileDropは、エクスプローラからファイルをドロップできるようにします。

説明　TFileDropは、Controlプロパティに指定したウィンドウハンドルを持つコントロールまたはフォーム、に Windows 95のエクスプローラからファイルをドロップできるようにします。

CHAP2¥DRGFILE.DPR

プロパティ

```
property Accept: Boolean;
```

説明　ファイルのドロップを受け入れるかどうかを指定します。

```
property Control: TWinControl;
```

説明　ファイルのドロップを受け入れたいコンポーネントを指定します。Controlが nilの場合は、TFileDropを配置したフォームが対象になります。

イベント

```
type
    TFileDropEvent = procedure (Sender: TObject; Num: Integer;
                      Files: TStrings; X, Y: Integer) of object;
property OnFileDrop: TFileDropEvent;
```

説明　OnFileDropイベントは、エクスプローラからControlプロパティで指定したコントロール（またはフォーム）にファイルがドロップされたときに発生します。引数Numにはファイルの数、Filesにはファイル名のリスト、X, Yにはドロップされた位置が渡されます。

TFileOperation

ユニット: NewCtrls

TObject・TPersistent・TComponent・TFileOperation

TFileOperationは、Shell APIを使ってファイルやディレクトリの削除、コピー、移動を実行します。

説明　TFileOperationは、複数のファイルやディレクトリをまとめて削除、コピー、移動したり名前を変更できます。ファイルを削除する場合、完全に削除する代わりにごみ箱に入れておくことができるため、ごみ箱を使って復元することもできます。また、処理中にアニメーションを表示したり、深い階層のディレクトリをまとめて処理できます。

CHAP3¥FILER.DPR

プロパティ

```
property FromFiles: TStrings;
```

説明　転送元あるいは削除するファイル名またはディレクトリ名のリストです。

```
type
    TFileOpFlag = (fofMultiDestFiles, fofConfirmMouse, fofSilent,
                   fofRenameOnCollision, fofNoConfirmation,
                   fofWantMappingHandle, fofAllowUndo, fofFilesOnly,
                   fofSimpleProgress, fofNoConfirmMkdir, fofNoErrorUI);
    TFileOpFlags = set of TFileOpFlag;
property Options: TFileOpFlags;
```

説明　ファイルを操作するときの動作を決定します。

値	意味
fofMultiDestFiles	ToFilesプロパティがディレクトリではなくFromFilesに対応するファイル名を保持していることをあらわします。
fofConfirmMouse	(実装されていません)
fofSilent	進行状況をあらわすダイアログを表示しません。
fofRenameOnCollision	対象となる名前がすでに存在する場合に「～のコピー」のような名前に置き換えます。
fofNoConfirmation	確認のダイアログを表示する代わりに「すべて、はい」として処理します。
fofWantMappingHandle	(FileOperationコンポーネントでは無効です)
fofAllowUndo	可能な場合は、取消のための情報を保存します。
fofFilesOnly	ワイルドカード(*.*)が指定されたとき、ファイルに対してだけ操作を実行します。
fofSimpleProgress	進行状況をあらわすダイアログを表示しますが、ファイル名は表示しません。
fofNoConfirmMkdir	ディレクトリを作成するときに確認しません。

```
property Title: string;
```

説明　進行状況ダイアログのためのタイトル文字列です。

```
property ToFiles: TStrings;
```

説明　転送先のファイル名またはディレクトリ名のリストです。削除のときは無視されます。

メソッド

```
function CopyFile(SrcFile, DstFile: TFileName): Bool;
```

説明　SrcFileからDstFileにファイルをコピーします。

```
function CopyFiles: Bool;
```

説明　FromFilesプロパティのファイルをToFilesプロパティにコピーします。

```
function DeleteFile(AFile: TFileName): Bool;
```

説明　AFileが示すファイルまたはディレクトリを削除します。

```
function DeleteFiles: Bool;
```

説明　FromFilesプロパティが示すファイルまたはディレクトリを削除します。

```
function MoveFile(SrcFile, DstFile: TFileName): Bool;
```

説明　SrcFileからDstFileにファイルを移動します。

```
function MoveFiles: Bool;
```

説明　FromFilesプロパティのファイルをToFilesプロパティに移動します。

```
function RenameFile(SrcFile, DstFile: TFileName): Bool;
```

説明　SrcFileが示すファイル名をDstFileに変更します。

```
function RenameFiles: Bool;
```

説明　FromFilesプロパティが示すファイル名をToFilesプロパティの内容に変更します。

TFontInfo

ユニット: NewCtrls

TObject ・ TPersistent ・ TComponent ・ TFontInfo

TFontInfoは、フォントの情報をレジストリに記録したり、復元します。

説明　TFontInfoは、与えられたフォントの情報をレジストリに記録し、次回の起動時に同じフォントの情報を使えるようにするものです。レジストリのキーは、KeyNameプロパティで設定しますが、これはすべてのTFontInfoコンポーネントで共通になります(TWinPlaceも共通のレジストリキーを使います)。

CHAP2¥USEWPLC.DPR

プロパティ

```
property EntryName: string;
```

説明　フォントの情報を保存するレジストリのエントリ名を指定します。KeyNameが指定するレジストリキーの中にこのプロパティで指定するエントリが作られて、フォントの情報が記録されます。

```
property KeyName: string;
```

説明　フォントの情報を保存するレジストリのキー名を指定します。このキーの中にEntryNameプロパティで指定するエントリが作られて、フォントの情報が記録されます。

メソッド

```
function ReadFont(const ADefault: TFont): TFont;
```

説明　レジストリからフォントの情報を読み出します。通常、フォームのOnCreateイベントハンドラで呼び出します。ADefaultは、レジストリにフォントの情報がない場合に使われるデフォルトフォントです。通常、代入先のフォントをそのまま指定します（例: Font := FontInfo1.ReadFont(Font);）。

```
procedure WriteFont(Value: TFont);
```

説明　フォントの情報をレジストリに書き出します。通常、フォームのOnDestroyイベントハンドラで呼び出します。

TIconTray

ユニット: SysCtrls

TObject・TPersistent・TComponent・TIconTray

TIconTrayは、Windows 95のトレイにアイコンを登録するためのコンポーネントです。

説明　TIconTrayは、Windows 95のタスクバーの右側にあるトレイにアイコンを登録します。KBP

Controlプロパティに指定されたコントロールにIME（かな漢字変換機能）を使って文字を入力したとき、OnCompositionStrイベントを発生させて使われた構成文字列（ヨミガナ）を渡します。マウスの左右のボタンのクリックに対応するポップアップメニューやイベントなどを定義できます。

CHAP2¥USETRAY.DPR

プロパティ

```
property ID: Integer;
```

説明　トレイに登録するアイコンを識別するための数値です。

```
property Icon: TIcon;
```

説明　トレイに登録するアイコンを指定します。

```
property HideMainForm: Boolean;
```

説明　アプリケーションを起動するときにメインフォームを表示しないようにするかどうかを指定します。HideMainFormがTrueのときは、Application.ShowMainFormがFalseになります。そうでない場合は、何もしません。

```
property RBtnPopupMenu: TPopupMenu;
```

説明　アイコン上でマウスの右ボタンがクリックされたときに呼び出されるポップアップメニューを指定します。

```
property LBtnPopupMenu: TPopupMenu;
```

説明　アイコン上でマウスの右ボタンがクリックされたときに呼び出されるポップアップメニューを指定します。

```
property ShowHint: Boolean;
```

説明　アイコン上にマウスカーソルを移動させたときに、ヒント文字列を表示するかどうかを指定します。

```
property Hint: string;
```

説明　アイコン上にマウスカーソルを移動させたときに表示するヒント文字列を指定します。ヒント文字列は最大63文字まで指定できます。また、ShowHintがTrueの場合にのみ有効です。

メソッド

```
procedure RegisterStartup(AnEntry: string);
```

説明　現在実行中のプログラムを、Windowsの起動時に実行させるようにレジストリに登録します。

```
procedure UnRegisterStartup(AnEntry: string);
```

説明　現在実行中のプログラムを、Windowsの起動時に実行させないようにレジストリの情報を削除します。

イベント

```
property OnClick: TNotifyEvent;
```

説明　アイコン上でマウスがクリックされたときに発生するイベントです。

```
property OnDblClick: TNotifyEvent;
```

説明　アイコン上でマウスがダブルクリックされたときに発生するイベントです。

```
property OnMouseDown: TMouseEvent;
```

説明　アイコン上でマウスのボタンが押されたときに発生するイベントです。

```
property OnMouseMove: TMouseMoveEvent;
```

説明　アイコン上でマウスカーソルが移動したときに発生するイベントです。

```
property OnMouseUp: TMouseEvent;
```

説明　アイコン上でマウスのボタンが離されたときに発生するイベントです。

TLine

ユニット: NewCtrls

TObject ・ TPersistent ・ TComponent ・ TControl ・ TGraphicControl ・ TLine

TLineは、直線を描画します。

説明　TLineは、長方形の対角線に直線を描画します。

CHAP6¥USELINE.DPR

プロパティ

```
property Pen: TPen;
```

説明　直線を描画するときに使うペンを設定します。

```
type
    TLineStart = (lsLeftTop, lsLeftBottom, lsRightBottom, lsRightTop);
property StartPos: TLineStart;
```

説明　直線を描画する始点（方向）を設定します。

値	意味
lsLeftTop	左上（から右下）
lsLeftBottom	左下（から右上）
lsRightBottom	右下（から左上）
lsRightTop	右上（から左下）

```
property X1: Integer;
```

説明　始点のX座標をあらわします。

```
property X2: Integer;
```

説明　終点のX座標をあらわします。

```
property Y1: Integer;
```

説明　始点のY座標をあらわします。

```
property Y2: Integer;
```

説明　終点のY座標をあらわします。

TMDIBack

ユニット: SysCtrls

TObject・TPersistent・TComponent・TMDIBack

TMDIBackは、MDIアプリケーションの背景に描画したい場合に配置するコンポーネントです。

説明　通常、MDIアプリケーションではMDIの背景に文字や図形を描画することはできません。FormStyleがMDIFormとなっているフォーム（MDIのメインフォーム）にTMDIBackを配置すると、フォームのOnPaintイベントで背景に描画したり、LabelやImageなどのコンポーネントを配置できるようになります。

また、TMDIBackは子ウィンドウの状態によって背景がスクロールされないよう、スクロールバーの表示を抑止します。このため、スクロールバーの表示を抑止するためだけにTMDIBackを配置することもできます。

TMDIBackは、MDIのメインフォームに配置するだけで効果を発揮するため、プロパティやメソッドはありません。

CHAP2¥MDIDRAW.DPR

TMMTimer

ユニット: NewCtrls

TObject・TPersistent・TComponent・TMMTimer

TMMTimerは、マルチメディアタイマーを使った精度の高いタイマーです。

説明　TTimerは、WindowsのマルチメディアタイマーAPIであるtimeSetEventを使ったタイマーコンポーネントです。TTimerと同じように使うことができますが、タイマーよりも精度が高くなります。

CHAP4¥STOPWATC.DPR

プロパティ

```
property ElapsedTime: Cardinal;
```

説明　ElapsedTimeは、コンポーネントが作成されてから、あるいはStartメソッドが呼び出されてから経過した時間をミリ描画単位で返します。簡単なストップウォッチ機能として利用できます。

```
property Enabled: Boolean;
```

説明　Enabledは、タイマーを有効にするか無効にするかを切り換えます。

```
property Interval: Cardinal;
```

説明　Intervalは、OnTimerイベントの発生間隔をミリ秒単位で示します。指定した時間が過ぎるとOnTimerイベントが発生します。Windowsでは、かならずしも厳密な時間が計測できない場合があります。

メソッド

```
procedure Start;
```

説明　TMMTimerを簡易ストップウォッチとして使う場合の、計測開始をあらわします。時間計測中は、ElapsedTimeを参照することでラップタイムをミリ秒単位で調べることができます。

```
procedure Stop;
```

説明　TMMTimerを簡易ストップウォッチとして使う場合の、計測終了をあらわします。時間計測後は、ElapsedTimeを参照することでかかった時間をミリ描画単位で調べることができます。

イベント

```
property OnTimer: TNotifyEvent;
```

説明　OnTimerイベントは、Intervalプロパティで指定した時間が経過すると発生します。イベントは連続的に発生するため1回で終了させる場合は、OnTimerイベントハンドラでEnabledプロパティにFalseを代入します。

TPageControlEx

ユニット: CtrlsEx

TObject・TPersistent・TComponent・TControl・TWinControl・TCustomTabControl・TPageControl・TPageControlEx

オーナー描画をサポートしたページコントロールです。

説明　TPageControlExは、プログラムでタブを描画できるオーナー描画をサポートしています。複数行や色付きなど、独自のスタイルのタブを使えます。また、TabPositionプロパティは、タブ位置を左右にも設定できるように拡張されています。

CHAP4¥TABPAGE.DPR

プロパティ

```
property Canvas: TCanvas;
```

説明　オーナー描画ページコントロールで、タブを描画する際に使うキャンバスです。

```
property OwnerDraw: Boolean;
```

説明　ページコントロールをオーナー描画スタイルにするかどうかを設定します。

```
type
    TTabPositionEx = (tpeTop, tpeBottom, tpeLeft, tpeRight);
property TabPosition: TTabPositionEx;
```

説明　タブの表示位置を指定します。タブは、上下だけでなく左右にも表示できます。

イベント

```
property OnDrawTab: TDrawItemEvent;
```

説明　オーナー描画ページコントロールで、タブを描画するために発生するイベントです。

TPort（TPortW）

ユニット: SysCtrls

TObject・TPort（TPortW）

I/Oポートに直接アクセスできるPort、PortW変数を提供します。

説明　通常、Windowsではアプリケーションが直接I/Oポートを制御するべきではありません。しかし、どうしてもI/Oポートを制御しなければならない場合は、Port（またはPortW）変数を使ってI/Oにアクセスできます。

I/Oポートへのアクセスは Port変数への代入や参照で表現できます。たとえば、PC-9800シリーズでビープ音を鳴らすためには Port[$37] := 6;とします。ビープ音を止めるためには Port[$37] := 7;とします。

Port、PortW変数は、SysCtrlsユニットであらかじめ提供されているためプログラムで生成する必要はありません。

CHAP3¥IOPORT.DPR

TPseudoApp

ユニット: SysCtrls

TObject・TPersistent・TComponent・TPseudoApp

TPseudoAppは、オブジェクトインスペクタを使ってApplicationオブジェクトのプロパティやイベントハンドラを設定するための疑似コンポーネントです。

説明　通常、Delphiのプログラミングで Applicationオブジェクトのプロパティやイベントハンドラを定義するためには、プログラムコードを記述します。TPseudoAppは、この手間をなくしオブジェクトインスペクタを使って Applicationオブジェクトのプロパティやイベントハンドラを設定できるようにするためのコンポーネントです。

TPseudoAppコンポーネントは、メインフォームに1つだけ配置できます。メインフォーム以外のフォームに配置された場合は、何もしません。また、メインフォームの OnCreateイベントハンドラなどで、Applicationのイベントハンドラが定義されている場合には、TPseudoAppの定義によって内容を置き換えません。

CHAP2¥USEPAPP.DPR

プロパティ

```
property HintColor: TColor;
property HintPause: Integer;
property HintShortPause: Integer;
property HintHidePause: Integer;
property ShowHint: Boolen;
property ShowMainForm: Boolean;
```

説明　これらは、TApplicationの同名のプロパティと同等の意味を持ちます。

イベント

```
property OnActivate: TNotifyEvent;
property OnDeactivate: TNotifyEvent;
property OnException: TExceptionEvent;
property OnIdle: TIdleEvent;
property OnHelp: THelpEvent;
property OnHint: TNotifyEvent;
property OnMessage: TMessageEvent;
property OnMinimize: TNotifyEvent;
property OnRestore: TNotifyEvent;
property OnShowHint: TShowHintEvent;
```

説明　これらは、TApplicationの同名のイベントと同等の意味を持ちます。

TQueryEx

ユニット: DBExCtls

TObject・TPersistent・TComponent・TDataSet・TBDEDataSet・TDBDataSet・TQuery・TQueryEx

ローカルテーブルの問い合わせ処理中にイベントを発生できるコンポーネントです。

説明　TQueryExコンポーネントでは、問い合わせ処理中にOnProgressイベントを発生させ、進行状況メッセージを表示させたり、問い合わせ処理を中断できるようにした問い合わせコンポーネントです。

問い合わせ処理中に、適当な間隔でOnProgressイベントが発生します。このイベントハンドラには、進行状況を示すMsgStr引数と処理を継続するかどうかを決めるためのAction引数が渡されます。これらの引数を使ってダイアログボックスにメッセージを表示したり、処理の継続をキャンセルできます。

CHAP5¥CHKDBEX.DPR

イベント

```
type
    TQueryProgressAction = (qpaContinue, qpaAbort);
    TQueryProgressEvent = procedure (Sender: TObject; MsgStr: string;
                          var Action: TQueryProgressAction) of object;
property OnProgress: TQueryProgressEvent;
```

説明　問い合わせ処理中に呼び出されるイベントです。MsgStrは、進行状況をあらわすメッセージ文字列が含まれており、"レコードは追加されました: <数値>"という形式をとります。Actionには、qpaCONTINUE（継続、デフォルト）、qpaABORT（中断）のいずれかを指定します。

なお、これらはBDEのデータベース処理から呼び出されるため、OnProgressイベントハンドラからデータベース処理を呼び出してはいけません。また、問い合わせ中は、Windowsメッセージも処理されないため、進行状況を表示するためにダイアログボックスを使う場合などは、メッセージを処理するためにApplication.ProcessMessagesを呼び出す必要があります。表示を更新するだけなら、コンポーネントのUpdateメソッドを呼び出してもかまいません。

TRichEditEx

ユニット: CtrlsEx

TObject・TPersistent・TComponent・TControl・TWinControl・TCustomEdit・TCustomMemo・TCustomRichEdit・TRichEdit・TRichEditEx

縦書きをサポートした書式付き編集コントロールです。

説明　TRichEditExコンポーネントでは、Verticalプロパティを Trueにすることで、縦書きで書式付きテキストを編集できます。また、文字のベースラインを変更することもできます。

CHAP4¥RTEDIT.DPR

プロパティ

```
property Vertical: Boolean;
```

説明　テキストを縦書きで編集するかどうかを指定します。縦書きで編集したテキストを書式付きテキスト（RTF）としてファイルに保存すると、他のRichEditコントロールで読み込む場合も縦書き表示になります。

メソッド

```
procedure SetBaseLine(AnOffset: Longint);
```

説明　選択されている範囲の文字列のベースライン（下端の位置）を設定します。AnOffsetに正の値を与えると上に、負の値を与えると下に文字列がずれます。改行幅には影響しないため、大きな値を与えると文字列は見えなくなります。

TScrollBarEx

ユニット: CtrlsEx

TObject・TPersistent・TComponent・TControl・TWinControl・TScrollBar・TScrollBarEx

つまみの幅を設定できるスクロールバーです。

説明　TScrollBarExは、つまみの幅を設定できるスクロールバーです。つまみの幅は、Pageプロパティで設定し、0の場合は標準のつまみの幅が使われます。Pageの最大値は、(Max - Min + 1)です。

CHAP4¥USESBAR.DPR

プロパティ

```
property Page: Integer;
```

説明　スクロールバーのつまみの幅を指定します。Pageの値は、Maxプロパティ と Minプロパティの差に対する相対値として評価されます。

TShellLink

ユニット: SysCtrls

TObject・TPersistent・TComponent・TShellLink

TShellLinkは、スタートアップメニューやデスクトップに、プログラムへのショートカットを作成します。

説明　TShellLinkは、アプリケーションのショートカットを作成するコンポーネントです。どの場所にショートカットを作成するかはLinkTypeプロパティで指定し、ショートカットの属性はその他のプロパティで指定します。

実際にショートカットを作成するときはCreateLinkメソッドを呼び出し、ショートカットを削除するときはRemoveLinkを呼び出します。

CHAP2¥WEEKWALL.DPR

プロパティ

```
property Arguments: string;
```

説明　ショートカットの属性のうち、アプリケーションに渡す引数を指定します。

```
property Description: string;
```

説明　ショートカットの名前を設定します。

```
property FileName: TFileName;
```

説明　ショートカットに割り当てる、アプリケーションのパス・ファイル名を指定します。

```
property HotKey: TShortCut;
```

説明　ショートカットに割り当てたプログラムをすばやく呼び出せるショートカットキーを指定します。

```
property IconIndex: Integer;
```

説明　ショートカットに割り当てるアイコンの位置を指定します。アイコンは、ショートカットに割り当てたプログラムから検索されます。

```
type
        TShellLinkType = (lnkDesktop, lnkReserved1, lnkPrograms,
            lnkControls, lnkPrinters, lnkPersonal, lnkFavorites,
            lnkStartup, lnkRecent, lnkSendTo, lnkBitBucket, lnkStartMenu,
            lnkReserved2, lnkReserved3, lnkReserved4, lnkReserved5,
            lnkDesktopDirectory, lnkDrives, lnkNetwork, lnkNetHood,
            lnkFonts, lnkTemplates, lnkCommonStartMenu, lnkCommonPrograms,
        lnkCommonStartup, lnkCommonDesktopDirectory, lnkAppData,
            lnkPrintHood);
property LinkType: TShellLinkType;
```

説明　ショートカットを割り当てる場所を指定します。

値	意味
lnkDesktop	Windowsのデスクトップ。
lnkReserved1	(予約済み)
lnkPrograms	ユーザーのプログラムグループを含むシステムディレクトリ
lnkControls	コントロールパネル
lnkPrinters	組み込み済みのプリンタを含む仮想フォルダ
lnkPersonal	ドキュメントの共通リポジトリとしてのファイルシステムディレクトリ
lnkFavorites	お気に入りのファイルを含むシステムディレクトリ
lnkStartup	スタートアッププログラムグループに対応するシステムディレクトリ
lnkRecent	最近使ったファイルを含むファイルシステムディレクトリ
lnkSendTo	送信メニュー項目を含むファイルシステムディレクトリ
lnkBitBucket	ごみ箱の中のシステムディレクトリ
lnkStartMenu	スタートメニュー項目を含むファイルシステムディレクトリ
lnkReserved2	(予約済み)
lnkReserved3	(予約済み)
lnkReserved4	(予約済み)
lnkReserved5	(予約済み)
lnkDesktopDirectory	デスクトップに直接記録されるファイルシステムディレクトリ
lnkDrives	マイコンピュータ上の仮想フォルダ
lnkNetwork	ネットワークでの隣りにあらわれるオブジェクトを含むファイルシステムディレクトリ
lnkNetHood	ネットワーク階層のトップレベルをあらわす仮想フォルダ
lnkFonts	フォントを含む仮想フォルダ
lnkTemplates	ドキュメントのためのファイルシステムディレクトリ
lnkCommonStartMenu	スタートメニュー項目を含む共通のファイルシステムディレクトリ
lnkCommonPrograms	ユーザーのプログラムグループを持つ共通のシステムディレクトリのフォルダ
lnkCommonStartup	スタートアッププログラムグループに対応する共通のシステムディレクトリ
lnkCommonDesktopDirectory	デスクトップに直接記録される共通のファイルシステムディレクトリ
lnkAppData	アプリケーションデータ
lnkPrintHood	プリンタ

```
property WindowState: TWindowState;
```

説明　ショートカットに割り当てるウィンドウ起動時の表示状態を指定します。

```
property WorkDir: string;
```

説明　ショートカットに割り当てる起動時の作業ディレクトリを指定します。

メソッド

```
procedure CreateLink;
```

説明　ショートカットを作成します。

```
procedure RemoveLink;
```

説明　ショートカットを削除します。LinkTypeおよびFileName以外のプロパティは意味を持ちません。

関連する手続き・関数

```
procedure CreateShellLink(FileName: TFileName; Desc: string;
    WorkDir: string; LinkType: TShellLinkType;
    WindowState: TWindowState; HotKey: TShortCut; IconIndex: Integer);
```

説明　TShellLinkコンポーネントを使わずに、直接ショートカットを作成します。引数の意味は、TShellLinkコンポーネントのプロパティ名とほぼ対応しています。

```
procedure RemoveShellLink(FileName: TFileName;
    LinkType: TShellLinkType);
```

説明　TShellLinkコンポーネントを使わずに、直接ショートカットを削除します。引数の意味は、TShellLinkコンポーネントのプロパティ名と対応しています。

TSingleInstance

ユニット: SysCtrls

TObject ・ TPersistent ・ TComponent ・ TSingleInstance

TSingleInstanceは、アプリケーションをひとつだけしか起動できないようにするコンポーネントです。

説明　通常、Delphiで作成したアプリケーションはいくつでも起動できます。TSingleInstanceコンポーネントを使うと、起動するアプリケーションをひとつだけに限定できます。また、DupInstActionプロパティを使って最初に起動したインスタンスをアクティブにしたり、OnDuplicateイベントハンドラで多重起動された場合の処理を定義することができます。

TSingleInstanceコンポーネントは、メインフォームに1つだけ配置できます。メインフ

ォーム以外のフォームに配置された場合は、正しく動作しないことがあります。

TSingleInstanceは、メインフォームが作成された後で多重起動をチェックするため、一瞬だけメインフォームが表示されてしまうことがあります。プロジェクトソースでCheckSingleInstance手続きを呼び出すことで、より早い段階で多重起動を検査できます。

CHAP2¥ONLYONE.DPR

プロパティ

```
type
    TDupInstAction = (diActivateFirst, diException, diTerminate);
property DupInstAction: TDupInstAction;
```

説明　多重起動された場合の処理を指定します。

値	意味
diActivateFirst	最初に起動したインスタンスをアクティブにします。
diException	多重起動を示す例外（EMultiAppInstance）を発生します。
diTerminate	何もせずに終了します。

```
property ErrorMessage: string;
```

説明　DupInstActionプロパティがdiExceptionに設定されているとき、多重起動において発生する例外で表示するメッセージとして使われます。

```
property UniqueName: string;
```

説明　アプリケーションを識別するためのユニークな文字列を指定します。デフォルトでは、乱数によって初期値が「Ident+<数値>」と設定されますが、アプリケーションにふさわしい名前に変更することをお勧めします。UniqueNameを使うことで、異なるアプリケーションの同時起動を抑止することもできます。

イベント

```
property OnDuplicate: TNotifyEvent;
```

説明　多重起動された場合に発生するイベントです。

関連する手続き

```
procedure CheckSingleInstance(AName: string; ActivateFirst: Boolean);
```

説明　コンポーネントを使わずに多重起動を調べる手続きです。ANameには、アプリケーションを識別するためのユニークな文字列を与えます。ActivateFirstは、多重起動の場合に最初に起動したインスタンスをアクティブにするかどうかを指定します。

通常、CheckSingleInstanceはプロジェクトソースの先頭で呼び出します。

```
program Project1;

uses
  Forms, SysCtrls,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  CheckSingleInstance('Test Application', True);
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end;
```

CheckSingleInstanceを使うことで、メインフォームが作成されるよりも前に多重起動を検査できます。

TSystemMenu

ユニット: NewCtrls

TObject・TPersistent・TComponent・TMenu・TPopupMenu・TSystemMenu

TSystemMenuは、システムメニューに新しい項目を追加します。

説明　TSystemMenuをダブルクリックすると、メニューデザイナが呼び出されてメニュー項目を編集します。TSystemMenuはTPopupMenuに似ていますが、システムメニュー（タイトルバーの左端をクリックして表示されるメニュー）に項目を追加します。また、既存の項目を表示しないように設定できます。TSystemMenuがメインフォームに配置されると、Windows 95のタスクバーに表示されるアプリケーション自身のメニューにも項目が追加されます。

CHAP2¥SMENU.DPR

プロパティ

```
type
    TSystemMenuCommand = (scSize, scMove, scMinimize, scMaximize,
                          scClose, scRestore);
    TSystemMenuCommands = set of TSystemMenuCommand;
property EnableCommands: TSystemMenuCommands;
```

説明　定義済みのシステムメニューから消去する項目を選びます。

値	意味
scSize	サイズ変更(S)
scMove	移動(M)
scMinimize	最小化(N)
scMaximize	最大化(X)
scClose	閉じる(C)
scRestore	元のサイズに戻す(R)

実行時に消去する項目を変更した場合は、Refreshメソッドを呼び出す必要があります。ただし、アプリケーション自身のメニューから削除した項目は復元できません。

メソッド

```
procedure Refresh;
```

説明　メニューを再構成します。プログラムの実行中にメニュー項目を変更したり、消去する項目を変更した場合に呼び出します。

TTabControlEx

ユニット: CtrlsEx

TObject・TPersistent・TComponent・TControl・TWinControl・TCustomTabControl・TTabControl・TTabControlEx

オーナー描画をサポートしたタブコントロールです。

説明　TTabControlExは、プログラムでタブを描画できるオーナー描画をサポートしています。複数行や色付きなど、独自のスタイルのタブを使えます。また、TabPositionプロパティは、タブ位置を左右にも設定できるように拡張されています。

CHAP4¥TABPAGE.DPR

プロパティ

```
property Canvas: TCanvas;
```

説明　オーナー描画タブコントロールで、タブを描画する際に使うキャンバスです。

```
property OwnerDraw: Boolean;
```

説明　タブコントロールをオーナー描画スタイルにするかどうかを設定します。

```
type
    TTabPositionEx = (tpeTop, tpeBottom, tpeLeft, tpeRight);
property TabPosition: TTabPositionEx;
```

説明　タブの表示位置を指定します。タブは、上下だけでなく左右にも表示できます。

イベント

```
property OnDrawTab: TDrawItemEvent;
```

説明　オーナー描画タブコントロールで、タブを描画するために発生するイベントです。

TTableEx

ユニット: DBExCtls

TObject・TPersistent・TComponent・TDataSet・TBDEDataSet・TDBDataSet・TTable・TTableEx

Paradox/dBASE特有の機能に対応したテーブルコンポーネントです。

説明　TTableExコンポーネントは、ParadoxおよびdBASE特有の機能に対応しています。

- Paradoxテーブルの自動リフレッシュ
- dBASE/Paradoxテーブルの圧縮
- 暗号化dBASEテーブルのサポート

Paradoxテーブルで、ネットワークでテーブルを共有している際にリモートでデータが更新されたかどうかを調べ、自動的にリフレッシュをかけることができます。

この機能を利用するためには、Timerコンポーネントと組み合わせて使います。TableExとTimerを配置し、TableExのTimerプロパティに配置したTimerコンポーネントの名前を指定します。これによって、タイマーのイベントが発生するたびに、自動的に

テーブルがリモートで更新されたかどうかを調べ、必要に応じてRefreshメソッドを呼び出します。スタンドアロン環境では自動リフレッシュは無効です。

また、dBASEテーブルではレコードを削除しても、実際の.DBFファイルでは削除フラグが付けられるだけで、完全には削除されません。Paradoxテーブルは、効率のためメモ項目などは削除されません。

PackTableメソッドを使うと、これらのテーブルから、削除すべきレコードを実際に削除し、テーブルを圧縮します。なお、別のアプリケーションで同じオープンしている場合は、圧縮できません。Delphiのフォーム上でTableExのActiveプロパティをTrueにしている場合も、オープンしていることになりますので注意してください。

TableExでは、暗号化されたdBASEテーブルをオープンすることもできます。AllowDBaseLoginプロパティをTrueにし、OnDBaseLoginイベントハンドラを定義することで、暗号化されたテーブルを扱えるようにします。

CHAP5¥PACKDB.DPR
CHAP5¥CHKDBEX.DPR

プロパティ

```
property AllowDBaseLogin: Boolean;
```

説明　暗号化されたdBASEテーブルを扱うかどうかを指定します。AllowDBaseLoginをTrueにすると、暗号化されたdBASEテーブルをオープンしようとするときにOnDBaseLoginイベントが発生します。ここで、適切なグループ名やユーザー名を返すことで、暗号化されたdBASEテーブルをオープンできます。

```
property Timer: TTimer;
```

説明　Paradoxテーブルの自動リフレッシュを行なうためのTimerコンポーネントを指定します。Timerコンポーネントが指定されていない場合は、Paradoxテーブルの自動リフレッシュは無効です。

メソッド

```
procedure PackTable;
```

説明　dBASEまたはParadoxテーブルの削除レコードを、完全にファイルから取り除き、ファイルサイズを圧縮します。

```
procedure ProcessMessages;
```

説明　自動リフレッシュを有効にしている場合、Application.ProcessMessagesの代わりに呼び出します。Timerプロパティを設定して自動リフレッシュを有効にしている場合、タイマーのイベントハンドラでリモート更新を調べます。テーブルや問い合わせのようなデータベースを処理するイベントハンドラでApplication.ProcessMessagesを呼び出すと、この処理中にタイマーイベントが発生して、さらにデータベース処理が呼び出されてしまう可能性があります（再入）。

TTableExのProcessMessagesは、Timerプロパティに割り当てられたタイマーの動作を禁止してからApplication.ProcessMessagesを呼び出すため、再入の問題を避けることができます。

```
property TableChanged: Boolean;
```

説明　テーブルがリモート更新されたかどうかを返します。直前の状態をたしかめるためには、TableChangedプロパティを参照する前にBDE APIのDbiCheckRefreshを呼び出します。

イベント

```
type
    TDBaseLoginEvent = procedure (Sender: TObject;
            var UserName, GroupName, UserPassword: string) of object;
property OnDBaseLogin: TDBaseLoginEvent;
```

説明　AllowDBaseLoginプロパティがTrueのとき、暗号化されたdBASEテーブルをオープンしようとすると発生するイベントです。このイベントハンドラで、オープンしようとするdBASEテーブルに対するUserName、GroupName、UserPasswordを設定すれば、暗号化されたdBASEテーブルをオープンできます。

ユーザー名やパスワードの入力には、LoginDialogを使うこともできます。

TWinPlace

ユニット: NewCtrls

TObject・TPersistent・TComponent・TWinPlace

TWinPlaceは、現在のフォームの位置や大きさをレジストリに記録したり、復元します。

説明　TWinPlaceは、現在のフォームの位置や大きさの情報をレジストリに記録し、次回の起動時に同じ位置と大きさに配置できるようにするものです。レジストリのキーは、KeyNameプロパティで設定しますが、これはすべてのTWinPlaceコンポーネントで共通

になります（TFontInfoも共通のレジストリキーを使います）。

CHAP2¥USEWPLC.DPR

プロパティ

```
property KeyName: string;
```

説明　フォームの位置を保存するレジストリのキー名を指定します。このキーの中にフォーム名のエントリが作られて、フォームの情報が記録されます。

メソッド

```
procedure ReadInfo;
```

説明　レジストリからフォームの情報を読み出します。通常、フォームのOnCreateイベントハンドラで呼び出します。

```
procedure WriteInfo;
```

説明　フォームの情報をレジストリに書き出します。通常、フォームのOnDestroyイベントハンドラで呼び出します。

クリックとダブルクリックの判別

ユニット: SysCtrls

クリックとダブルクリックを区別して処理したい場合に使う関数です。

説明　通常、Windowsではダブルクリックする場合でも、必ずその前にクリックイベントが発生します。このため、（単一の）クリックとダブルクリックで異なる処理をしたい場合には、単純にOnClickとOnDblClickを使うことはできません。

MarkDoubleClick手続きとCheckDoubleClick関数を使うことで、クリックとダブルクリックで異なる処理を実行できます。この2つの手続きと関数は、必ず対で使用してください。

MarkDoubleClickは、コントロール（またはフォーム）のOnClickイベントハンドラで呼び出します。OnClickイベントハンドラでは、これ以外の処理はしません。CheckDoubleClickは、コントロールのOnDblClickイベントハンドラで呼び出します。CheckDoubleClickがTrueを返す場合はダブルクリックのための処理を、Falseを返す場

合はクリックのための処理を実行します。

クリックとダブルクリックを区別するためには、最初のクリックから一定期間に2回目のクリックがないことを確認しなければなりません。このため、クリックの処理でも、一定期間は処理を開始できないことに注意してください。

CHAP4¥DBLCLK.DPR

関数と手続き

```
procedure MarkDoubleClick(ACtrl: TObject);
```

説明 OnClickイベントハンドラで呼び出します。内部では、クリックとダブルクリックを調べるためのコントロールを設定し、内部でタイマーを設定します。

```
function CheckDoubleClick(ACtrl: TObject): Boolean;
```

説明 OnDblClickイベントハンドラで呼び出します。OnDblClickイベントは、ダブルクリックされた場合、および最初のクリックから一定時間に2回目のクリックがなかった場合に発生します。CheckDoubleClickを呼び出すことで、どちらに該当するかを判別できます。Trueを返す場合はダブルクリックの処理を、Falseを返す場合はクリックの処理を行ないます。

月・曜日名の設定

ユニット: SysCtrls

英単語の月名や曜日名を使いたい場合に呼び出します。

説明 Delphi 3では、FormatDateTimeなどでaaaaやmmmmなどの書式が使われると、システムの設定に合わせて月名や曜日名を返します。このため、日本語Windowsでは日本語の月名（1月、2月など）や曜日名（日曜日、月曜日など）が返されます。

これらの名前は、ShortMonthNamesやLongDayNamesなどのグローバル変数に登録されているため、InitMonthDayNamesを呼び出すことで英単語の月名（January、Februaryなど）や曜日名（Sunday、Mondayなど）を使えるように再初期化できます。

CHAP3¥DAYNAMES.DPR

手続き

```
procedure InitMonthDayNames;
```

説明　月名や曜日名の内部データを英単語に置き換えます。

```
procedure RestoreMonthDayNames;
```

説明　月名や曜日名の内部データを初期状態に戻します。

文字判別関数

ユニット: SysCtrls

C/C++の文字判別関数に対応する関数群です。

説明　C/C++で使われている、一般的な文字判別関数に対応する関数群です。

CHAP3¥CHKCHAR.DPR

関数

```
function isalnum(c: Char): ByteBool;          { 英数字 }
function isalpha(c: Char): ByteBool;          { 英字 }
function isascii(c: Char): ByteBool;          { ASCII 文字 }
function iscntrl(c: Char): ByteBool;          { 制御文字 }
function isdigit(c: Char): ByteBool;          { 数字 }
function isgraph(c: Char): ByteBool;          { 表示文字 }
function islower(c: Char): ByteBool;          { 英小文字 }
function isprint(c: Char): ByteBool;          { 印刷可能文字 }
function ispunct(c: Char): ByteBool;          { 区切り文字 }
function isspace(c: Char): ByteBool;          { 空白文字 }
function isupper(c: Char): ByteBool;          { 英大文字 }
function isxdigit(c: Char): ByteBool;         { 16進文字 }
function iskanji(c: Char): Boolean; { 漢字の1バイト目 }
function iskanji2(c: Char): Boolean;          { 漢字の2バイト目 }
```

説明　それぞれの関数は、引き数に与えられた文字が右側にコメントされている文字であればTrueを、そうでなければFalseを返します。

CHAP3¥CHKCHAR.DPR

INDEX

アルファベット

A・C

D・E

F・I・M・N

P・R・W・Z

五十音

あ

か

著者紹介

大野元久（おおのもとひさ）

1989年、名古屋工業大学修士号取得。同年、当時ボーランドの言語製品を扱っていたマイクロソフトウェアアソシエイツ（MSA）に入社。ボーランド言語のテクニカルサポートなどを経て、現在ボーランド株式会社にてマーケティングを担当。パソコン通信でのハンドルは「Oh!No!」。

Delphi 3 Q&A 150選

1997年9月29日 初版第1刷発行

監　修　ボーランド株式会社
著　者　大野元久
発行者　中村満
発行所　株式会社ビレッジセンター出版局
〒101
東京都千代田区神田神保町3-2
サンライトビル6F
TEL　03-3221-3520
FAX　03-3221-3528
印刷所　株式会社サムエンタープライズ

ISBN4-89436-105-1　Printed in Japan

株式会社ビレッジセンター出版局

■本書の構成

本書では、Delphiを活用するために役立つ150個のQ&Aを取り上げています。

■付録ディスク

本書のすべてのQ&Aをカバーしたヘルプファイルが収録されています。
また、100個以上の完結したサンプルプログラムや25個以上の便利なコンポーネントがソースコード付きで収録されています。

- テーブルの印刷やプレビュー機能
- メタファイルの作成
- バルーンヒント表示
- 独自の配色を使った文字列グリッド
- テキストデータとテーブルの変換
- VBやC/C++データの利用
- 色指定や複数行に対応したボタンやタブ
- ヨミガナを取得するコンポーネント
- 精度の高いタイマー
- ショートカットの作成
- システムメニューの拡張

ISBN4-89436-105-1　C3055　¥2000E　定価(本体2,000円＋税)