3.5″2HD(1.44MB)
ディスク付

Delphi 2.0

デルファイ

Q&A 120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（デルファイ） 2.0

Q&A 120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 2.0
（テルファイ）

Q&A 120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 2.0 Q&A 120選」

Delphiは、最初のバージョンが発売されて1年あまりが過ぎ、32ビット版のDelphi 2.0の発売によって迅速なアプリケーション開発ツールとしての地位を不動のものにしました。Delphi 2.0は多くの機能拡張が施されており、いっそう高速化されたコンパイラとともに非常に高い評価を得ています。

拡張された機能にはフォームの継承やオートメーションのサポート、データベースグリッドの拡張、レポートコンポーネント、通貨型、長い文字列などがあり、従来要望が多かったものがかなり網羅されています。

こうした機能拡張やWindows 3.1からWindows95への仕様変更により、「Delphi Q&A 120選」に掲載したQ&Aの中にはDelphi 2.0では適当でなかったり無用なものになってしまったものがあります。たとえば、Delphi 1.0で64KBを越えるメモリを扱うためにTMemoryStreamを使っていましたが、Delphi 2.0では理論的に2GBまでの膨大なメモリを直接処理できます。また、255文字を越える文字列も簡単に扱えるようになりました。

そこで、本書では32ビット版Delphi 2.0とWindows95に的を絞りQ&Aを再編成しました。もちろん、不要になったトピックを取り除くだけでなく、従来の項目やサンプルプログラムについても見直し、さらに新しい項目を追加しています。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。

私事ですが、編集を担当されたビレッジセンターの多田尚弘氏には、遅い原稿をいつまでも待っていただきました。氏の熱意がなければ本書は刊行に至らなかったと思います。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 2.0 Q&A 120選」利用法

本書には、Delphi 2.0に関する質問と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 2.0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。

また、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における日本語版Delphi 2.0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、動作についての確認は行なっておりません。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

書体・記号などの表記は、原則としてDelphiユーザーズガイドに準じます。

回答欄左上部のフェイスマークは問題の難易度を表わしています。


Q＆Aが容易に理解できることを表わしています。


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わしています。


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わしています。


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わしています。

難易度マークは、あくまでもめやすとしてご利用ください。Q＆Aが非常に高度な知識を必要とする場合でも、提示されているプログラムを利用するのはさほど難しくない場合があります。

プログラムファイルに関して

質問に対する回答で、付録ディスク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FD」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、付録ディスクの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。

CONTENTS

第1章 統合開発環境

第2章 プログラミング

第3章 アプリケーション／フォーム

第4章 コンポーネント

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。同じ名前では保存できないのでしょうか。

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)**|**ﾌﾟﾛｼﾞｪｸﾄ 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 2.0のあるディレクトリがデフォルトになっています。[ﾌﾟﾛｼﾞｪｸﾄ(P)]-[ｵﾌﾟｼｮﾝ(O)]-[ﾃﾞｨﾚｸﾄﾘ/条件]で[出力ﾃﾞｨﾚｸﾄﾘ(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 2.0]グループのDelphi 2.0ショートカットで右クリックし、[ﾌﾟﾛﾊﾟﾃｨ(R)]を選びます。ここで[ｼｮｰﾄｶｯﾄ]ページの[作業ﾌｫﾙﾀﾞ(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。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 2.0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。

[ﾌﾟﾛｼﾞｪｸﾄ(P)]-[ｵﾌﾟｼｮﾝ(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。[ﾃﾞｨﾚｸﾄﾘ/条件]ページの[出力ﾃﾞｨﾚｸﾄﾘ(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。

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。

フォームや配置したコンポーネントを設計し、修正している間に、不用意にプロパティやイベントハンドラを変更してしまい、原因を突き止めにくくなってしまうことがあります。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行ずつ追いかけるということができないためです。

Delphi 2.0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。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。ただし、そのフォームから継承しているフォームがあればテキスト表示に切り替えることはできません。この場合は、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。

フォームの情報は、次のようなテキストに展開されます。これを編集した上でスピードメニューの［フォームとして表示(V)］を選ぶと、ビジュアルなフォームとして表示され変更された内容が反映されます。

```
object Form1: TForm1
  Left = 200
  Top = 105
  Width = 435
  Height = 300
  Caption = 'Form1'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'ＭＳ Ｐゴシック'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 16
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。

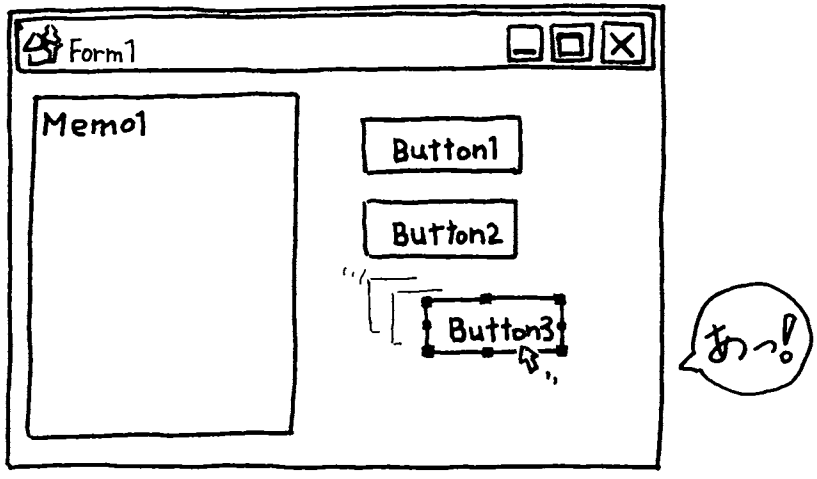
```
object AboutBox: TAboutBox
  Left = 243
  Top = 108
  ActiveControl = OKButton
  BorderStyle = bsDialog
  Caption = 'バージョン情報'
  ClientHeight = 214
  ClientWidth = 294
  Font.Color = clBlack
  Font.Height = -12
  Font.Name = 'ＭＳ Ｐゴシック'
  Font.Style = []
  Position = poScreenCenter
  PixelsPerInch = 96
  TextHeight = 12
  object Panel1: TPanel
    Left = 8
    Top = 8
    Width = 277
    Height = 158
    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ことがあるかもしれません。

こういった場合には、[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。

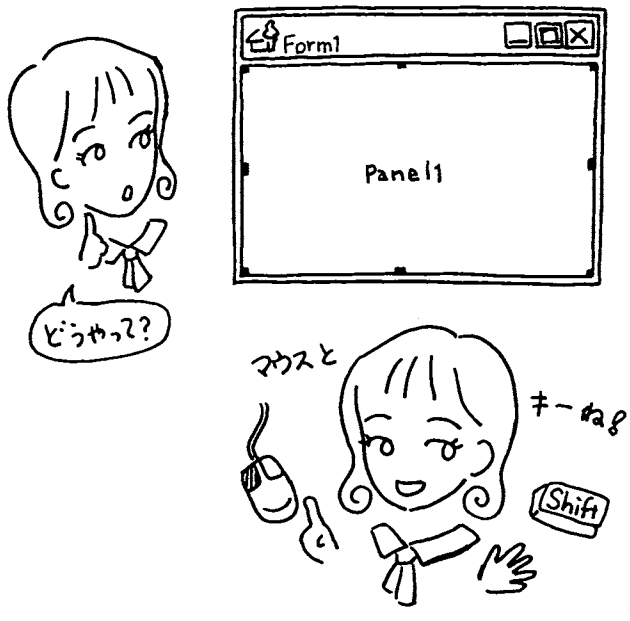


Q. フォームにパネルを配置してAlignプロパティをalClientにしているため、フォームをクリックする場所が隠れています。フォームのプロパティを変更するためには、どうすればよいでしょうか。また、フォーム上のコンポーネントをドラッグの範囲指定で選択するように、パネル上のコンポーネントを選択することはできませんか。

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。

フォーム上でひと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. 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200KB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ひとつの特長ですが、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、フォームやそれに関わるプログラムコードは実行ファイルの中に埋め込まれています。

このため、Delphiのビジュアルプログラミングを使って開発したものは、ごく単純なプログラムでも内部でのさまざまな関連性によって200KB程度の大きさになります。なお、Delphiに組み込まれているObject Pascalには、スマートリンクの機能があるためまったく使われていない関数や手続きは実行ファイルには含まれていません。また、フォームをひとつ増やすたびに200KB増加するわけではありません。

どうしても小さいプログラムを必要とする場合は、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.
```

付録FD　CHAP1¥SMALLAPP.DPR

Q. コンポーネントを作成して、[ｺﾝﾎﾟｰﾈﾝﾄ(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)|Image Editor] 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. 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が、[ﾂｰﾙ(T)|ｵﾌﾟｼｮﾝ(O)|設定] で [ｺﾝﾊﾟｲﾙ状況の表示(C)] をチェックしておけ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Events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。

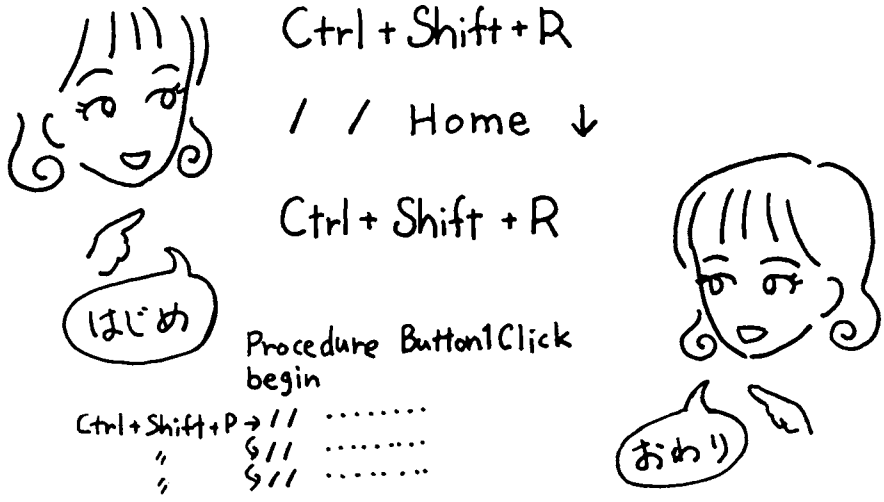
Q. コードエディタで、同じキー操作を繰返し実行するための機能はありませんか。

〔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〕+〔R〕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、プログラミングの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〕+〔0〕のようなキー操作で直接しおりの場所に移動できます。

ヘルプ　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逆アセンブルビューを表示できます。具体的にはHKEY_CURRENT_USER¥Software¥Borland¥Delphi¥2.0¥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¥2.0');
  Reg.WriteInteger('Debugging', 'EnableCPU', 1);
  Reg.Free;
end;
```

注意 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。

付録FD CHAP1¥CPUVIEW.DPR

第2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            { エラー }
  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。

付録FD　CHAP2¥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。

付録FD　CHAP2¥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。

付録FD　CHAP2¥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GetForegroundWindowを使います。

付録FD　CHAP2¥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。

付録FD　CHAP2¥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、いずれかひと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ひとつしか選べません。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。

付録FD　CHAP2¥RBUTTONS.DPR

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;
```

付録FD　CHAP2¥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。

付録FD　CHAP2¥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 = (49,0,22,94,98,30,0,30,76,94);
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;
```

付録FD　CHAP2¥MKMETAS.DPR

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、Windowsに付属する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すことができます。

```
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);
```

CreateProces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 2.0 にし、}
  { メモ帳 (NOTEPAD.EXE) を起動する }
  ShellExecute(Handle, 'OPEN', 'NOTEPAD.EXE', '',
               'C:¥Program Files¥Borland¥Delphi 2.0', 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;
```

付録FD　CHAP2¥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。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

付録FD CHAP2¥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。

付録FD CHAP2¥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。

付録FD　CHAP2¥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;
```

付録FD　CHAP2¥PRTEXT.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。

フォームやPaintBoxコンポーネントなどで使われているCanvasには、Drawというメソッドがあり、ビットマップやアイコンなどのグラフィックを簡単に描画できるようになっています。これは、ビットマップのためのキャンバスがディスプレイ互換のデバイスコンテキストになっているためです。ディスプレイコンテキストと互換性のないプリンタデバイス（モノクロプリンタなど）では、正常にビットマップイメージを転送できないため、Drawでは正常に描画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。

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）を、サポートされていなければFalse（偽）になります。

```
procedure PrintBitmap(X, Y: Integer; Bitmap: TBitmap);
var
  Info: PBitmapInfo;
  InfoSize: Integer;
  Image: Pointer;
  ImageSize: DWord;
begin
  with Bitmap do
  begin
    { DIB の大きさを取得 }
    GetDIBSizes(Handle, InfoSize, ImageSize);
    Info := MemAlloc(InfoSize);
    try
      Image := MemAlloc(ImageSize);
      try
        { DIB データを取得 }
        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;
```

付録FD　CHAP2¥PRTEST.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 2.0にはWinCrtはないのでしょうか。

Delphi 2.0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)|リンカ]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.
```

付録FD　CHAP2¥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 2.0では使えません。しかし、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。

以下のプログラムは、Delphi 1.0のPort配列と同じ動作をするクラスを提供します。このユニットをプロジェクトに追加すれば、直接I/Oを制御できます。

付録FD　CHAP2¥PORTS.PAS

```
unit Ports;

interface

type
  TPort = class
    function GetData(Index: Word): Byte;
    procedure SetData(Index: Word; Value: Byte);
  public
    property Data[Index: Word]: Byte
             read GetData write SetData; default;
  end;

  TPortW = class
    function GetData(Index: Word): Word;
    procedure SetData(Index: Word; Value: Word);
  public
    property Data[Index: Word]: Word
             read GetData write SetData; default;
  end;

var
  Port: TPort;
  PortW: TPortW;

implementation

function TPort.GetData(Index: Word): Byte;
begin
  asm
     mov dx, Index
     in al, dx
  end;
end;
```

```
procedure TPort.SetData(Index: Word; Value: Byte);
begin
  asm
     mov dx, Index
     mov al, Value
     out dx, al
  end;
end;

function TPortW.GetData(Index: Word): Word;
begin
  asm
     mov dx, Index
     in ax, dx
  end;
end;

procedure TPortW.SetData(Index: Word; Value: Word);
begin
  asm
     mov dx, Index
     mov ax, Value
     out dx, ax
  end;
end;

initialization
  Port := TPort.Create;
  PortW := TPortW.Create;

finalization
  PortW.Free;
  Port.Free;
end.
```

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。

付録FD　CHAP2¥IOPORT.DPR

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。以下にBorland C++と互換性を持つ文字列判定関数のプログラム例を示します。

付録FD　CHAP2¥CTYPE.PAS

```
unit Ctype;

{ Ctype ユニット : 文字列判別関数          }
{   Copyright (c) 1996 Borland Intl, all rights reserved.  }

interface

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 }

implementation

const
     _IS_SPACE = $01;
     _IS_DIGIT = $02;
     _IS_UPPER = $04;
     _IS_LOWER = $08;
     _IS_HEX   = $10;
     _IS_CNTRL = $20;
     _IS_PUNCT = $40;
```

```
     { Borland C++ と同等の判別用テーブル }
     _Ctype: array[Char] of Byte = (
     $20,$20,$20,$20,$20,$20,$20,$20,$20,$21,$21,$21,$21,$21,$20,$20,
     $20,$20,$20,$20,$20,$20,$20,$20,$20,$20,$20,$20,$20,$20,$20,$20,
     $01,$40,$40,$40,$40,$40,$40,$40,$40,$40,$40,$40,$40,$40,$40,$40,
     $02,$02,$02,$02,$02,$02,$02,$02,$02,$02,$40,$40,$40,$40,$40,$40,
     $40,$14,$14,$14,$14,$14,$14,$04,$04,$04,$04,$04,$04,$04,$04,$04,
     $04,$04,$04,$04,$04,$04,$04,$04,$04,$04,$04,$40,$40,$40,$40,$40,
     $40,$18,$18,$18,$18,$18,$18,$08,$08,$08,$08,$08,$08,$08,$08,$08,
     $08,$08,$08,$08,$08,$08,$08,$08,$08,$08,$08,$40,$40,$40,$40,$20,
     $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,
     $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,
     $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,
     $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,
     $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,
     $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,
     $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,
     $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
     );

function isalnum(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c]
                     and (_IS_DIGIT or _IS_UPPER or _IS_LOWER));
end;

function isalpha(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c] and (_IS_UPPER or _IS_LOWER));
end;

function isascii(c: Char): ByteBool;
begin
  Result := ByteBool(Ord(c) < 128);
end;

function iscntrl(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c] and _IS_CNTRL);
end;

function isdigit(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c] and _IS_DIGIT);
end;

function isgraph(c: Char): ByteBool;
begin
  Result := ByteBool(($21 <= Ord(c)) and (Ord(c) <= $7e));
end;
```

```
function islower(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c] and _IS_LOWER);
end;

function isprint(c: Char): ByteBool;
begin
  Result := ByteBool(($20 <= Ord(c)) and (Ord(c) <= $7e));
end;

function ispunct(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c] and _IS_PUNCT);
end;

function isspace(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c] and _IS_SPACE);
end;

function isupper(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c] and _IS_UPPER);
end;

function isxdigit(c: Char): ByteBool;
begin
  Result := ByteBool(_Ctype[c] and (_IS_DIGIT or _IS_HEX));
end;

function iskanji(c: Char): Boolean;
begin
  Result := (($81 <= Ord(c)) and (Ord(c) <= $9F))
        or (($E0 <= Ord(c)) and (Ord(c) <= $FC));
end;

function iskanji2(c: Char): Boolean;
begin
  Result := (($40 <= Ord(c)) and (Ord(c) <= $7F))
        or (($80 <= Ord(c)) and (Ord(c) <= $FC));
end;

end.
```

付録FD　CHAP2¥CHKCHAR.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
    with LogFont do
    begin
      lfHeight := 16;              { フォントの大きさ }
      lfWidth := 0;
      lfEscapement := Angle;       { 角度(0..3600) = (0..2π) }
      lfOrientation := 0;
      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 }
      lfCharSet := DEFAULT_CHARSET;
      lfOutPrecision := OUT_DEFAULT_PRECIS;
      lfClipPrecision := CLIP_DEFAULT_PRECIS;
      lfQuality := DEFAULT_QUALITY;
      lfPitchAndFamily := DEFAULT_PITCH;
      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;
```

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。

付録FD　CHAP2¥ANGLESTR.DPR

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、これらは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。

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; export;
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 }
end;

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

付録FD　CHAP2¥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, '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;
```

ヘルプ　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「入出力ルーチン」を参照してください。型なしファイルについては「型なしファイルルーチン」、テキストファイルについては「テキストファイルルーチン」を参照してください。

ヘルプ　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。

付録FD　CHAP2¥IDENTFIL.DPR
CHAP2¥VIEWINI.DPR

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;
```

付録FD　CHAP2¥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 2.0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 2.0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;
```

付録FD　CHAP2¥WINDIR.DPR

第3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. ひと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「ﾌｫｰﾑ'XXX' はUSESﾘｽﾄに無い 'YYY' ﾕﾆｯﾄで定義されているﾌｫｰﾑ'ZZZ' を参照しています。このﾕﾆｯﾄを追加してよいですか？」と表示されます。これは、どんな意味なのでしょうか。

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 2.0では、指定されていないフォームユニットが同じプロジェクトの別のユニットにある場合には、自動的に必要なユニット名を探し出してこのような警告を表示します。ここで［はい(Y)］を選べば、自動的に必要なユニットを探してUses節を追加します。このときはinterface部ではなくimplementation部に追加されます。なお、［ﾌｧｲﾙ(F)|ﾕﾆｯﾄを使う(U)］でも同じことができます。

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。

ユニットオブジェクトファイルには型だけでなくコンパイルされた情報そのものが含まれています。これは、C/C++処理系で使われているプリコンパイルヘッダのようなものとみなすこともできます。ユニットは、Delphiのコンパイル速度が高速な理由のひとつです。

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。

ヘルプ　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。

付録FD　CHAP3¥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。

Delphi 1.0で作成したアプリケーションが呼び出されるとき、以前にそのアプリケーションが起動されているかどうかを調べるにhPrevInstというグローバル変数を調べることができましたが、これはWin32では利用できません。

Win32では、アプリケーションのメインフォームが既に作成されているかどうかを調べることで、二重起動を抑止できます。[表示(V)|プロジェクト ソース(S)]で表示されるプロジェクトソースを以下のように変更してください。

```
program Project1;

uses
  Form1,
  Windows, { Windows API を使うために必要 }
  Unit1 in 'UNIT1.PAS' {Form1};

{$R *.RES}

var
  MainWnd: THandle;

begin
  MainWnd := FindWindow('TForm1', nil);
  if MainWnd <> 0 then
  begin
    { 既にメインフォームが作成されていれば、}
    { そのウィンドウをアクティブにして終了する }
    SetForegroundWindow(MainWnd);
    Exit;
  end;

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

この方法では、Delphiでプロジェクトを開いているときもフォームが作成されているため、開発中に実行できなくなります。よりよい方法としてミューテックスというオブジェクトを作成する方法があります。アプリケーション固有の名前を付けたミューテックスオブジェクトが既に存在する場合はアプリケーションを終了し、存在しなければアプリケーションを実行するようにします。

```
  ...
const
  UniqueName = 'MutexObjectForApp';

var
  hMutex: THandle;

begin
  hMutex := OpenMutex(MUTEX_ALL_ACCESS, False, UniqueName);
  if hMutex <> 0 then
  begin
    CloseHandle(hMutex);
    Exit;
  end;
  hMutex := CreateMutex(nil, False, UniqueName);
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
  ReleaseMutex(hMutex);
end.
```

重要　統合開発環境からは、ひとつのアプリケーションしか実行できません。

付録FD　CHAP3¥ONLYONE.DPR

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。

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。

ヘルプ　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。

付録FD　CHAP3¥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付きで渡されます。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数として解釈されます。たとえば、SAMPLE.EXEに「one "two three" four」という引数を渡した場合は、次のようになります。

```
ParamCount:  4
ParamStr(0):  C:¥WORK¥SAMPLE.EXE
ParamStr(1):  one
ParamStr(2):  two three
ParamStr(3):  four
```

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に代入できます。

付録FD　CHAP3¥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［ﾌﾟﾛｼﾞｪｸﾄ(P)｜ｵﾌﾟｼｮﾝ(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｜ﾌﾟﾛｼﾞｪｸﾄ 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;
```

付録FD　CHAP3¥DYNFORM.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。

付録FD　CHAP3¥MDIPROG.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(O)｜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。

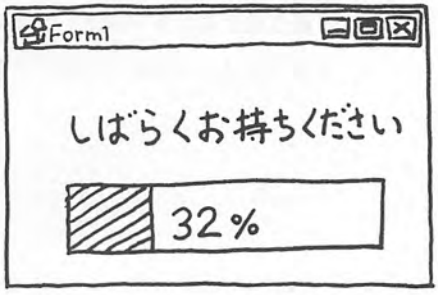
前述と同様にオープニングフォームにGaugeコンポーネントを配置しておき、残りのフォームの数をMaxValueプロパティに合わせておき、CreateFormでフォームを作成するごとにGaugeコンポーネントのProgressに値を設定しておけば、フォームが作成されるたびにゲージを更新できます。

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。

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。

```
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.
```

付録FD　CHAP3¥PROGRESS.DPR
CHAP3¥SPLASH.DPR



Q. 作成したプログラムをグループに登録し、[ﾌﾟﾛﾊﾟﾃｨ(R)]-[ｼｮｰﾄｶｯﾄ]で[実行時の大きさ(R)]を最大化にしておいたのですが、最大化されずに通常の状態で起動してしまいます。

フォームは、アプリケーションに渡される表示状態の設定に関係なく、WindowStateプロパティに設定した内容に基づいて表示されます。渡される表示状態はCmdShowというグローバル変数で参照でき、フォームが作成された直後にこの変数をもとにWindowStateプロパティを変更することができます。ごく簡単に処理するためには、プロジェクトソースを変更して次のように記述します。

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);

  { メインフォームを作成した後、表示状態をあらわすグローバル変数 }
  { CmdShow をもとにして、WindowState プロパティを変更する }
  with Application.MainForm do
    case CmdShow of
    2,6,7: WindowState := wsMinimized;
    3: WindowState := wsMaximized;
    else WindowState := wsNormal;
    end;

  Application.Run;
end.
```

この場合、最小化されたプログラムはタスクバーに入らず画面の左下に小さく表示されることになります。最小化が指定された場合にプログラムをタスクバーに納めるためには、次のようなやや面倒な記述が必要です。

```
type
  TAppHelper = class
    procedure RestoreMainForm(Sender: TObject);
  end;

procedure TAppHelper.RestoreMainForm(Sender: TObject);
begin
  Application.MainForm.Show;
end;

var
  AppHelper: TAppHelper;

begin
  AppHelper := TAppHelper.Create;
  Application.OnRestore := AppHelper.RestoreMainForm;

  Application.Initialize;
  Application.CreateForm(TForm1, Form1);

  { メインフォームを作成した後、表示状態をあらわすグローバル変数 }
  { CmdShow をもとにして、WindowState プロパティを変更する }
  with Application.MainForm do
    case CmdShow of
    2,6,7:
      begin
        Application.ShowMainForm := False;
        Application.Minimize;
      end;
    3: WindowState := wsMaximized;
    else WindowState := wsNormal;
    end;

  Application.Run;
end.
```

付録FD　CHAP3¥CMDSHOWP.DPR

Q. あるマシンで作成したフォームを別のマシンで実行すると、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いてしまいました。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。

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。

付録FD　CHAP3¥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。

付録FD　CHAP3¥SCRLPOS.DPR

Q. レジストリを操作したいのですが、TRegistryの説明が少なく使い方がわかりません。

Delphi 2.0でレジストリを操作したい場合は、TRegIniFileを使います。TRegIniFileは、初期化ファイル（.INI）の代わりにレジストリを使うという点以外はTIniFileと同じメソッドやプロパティを持っています。たとえば、次のように記述すればHKEY_CURRENT_USERにSoftware¥SampleAppキーが作成され、Default¥Leftに10が、Default¥Topに20が書き込まれます。値を参照する方法もTIniFileと同じです。

```
var
  Ini: TRegIniFile;
begin
  Ini := TRegIniFile.Create('Software¥SampleApp');
  Ini.WriteInteger('Default', 'Left', 10);
  Ini.WriteInteger('Default', 'Top', 20);
  Ini.Free;
end;
```

TRegIniFileを使う際はRegistryユニットをuses節に追加してください。

付録FD　CHAP3¥USEWPLC.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。

Delphiには、レジストリを扱うためのTRegIniFileというクラスが用意されています。これは、Delphi 1.0で初期化ファイル（.INI）を扱うためのTIniFileに相当するもので、初期化ファイルと同じようにレジストリを利用できる便利なクラスです。また、ウィンドウの状態を取得・設定するためには、GetWindowPlacement、SetWindowPlacementというWindows APIがあります。

これらを組み合わせたプログラムを以下に示します。

付録FD　CHAP3¥WINPLACE.PAS

```
unit Winplace;

interface

uses
  Windows, Messages, SysUtils, Classes, Forms, Registry;

{
  このユニットでは、以下の 4 つの手続きが提供されています。

  ReadFormPlacement: フォーム位置の読み込み
  ReadFormPlacementSection: フォーム位置の読み込み（キー名指定付）
  WriteFormPlacement: フォーム位置の書き込み
  WriteFormPlacementSection: フォーム位置の書き込み（キー名指定付）

  読み込み手続きは、フォームの OnCreate イベントハンドラに書きます。
  procedure TForm1.FormCreate(Sender: TObject);
  begin
    ReadFormPlacement(Self);
  end;

  書き込み手続きは、フォームの OnDestroy イベントハンドラに書きます。
  procedure TForm1.FormDestroy(Sender: TObject);
  begin
    WriteFormPlacement(Self);
  end;

  レジストリキーには、HKEY_CURRENT_USER¥Software¥DelphiApp に
  アプリケーションの実行ファイル名（拡張子を取り除いたもの）を
  追加して使います。個々のフォームの状態は、フォームのクラス名を
  使います。通常は、フォームごとにクラス名が異なるため問題はありませんが、
  MDI の子フォームなどで同じクラス名から複数のフォームを作成する場合は、
  キー名指定つきの手続きを使ってください。
}
```

```
procedure ReadFormPlacement(Form: TForm);
procedure ReadFormPlacementSection(Form: TForm;
                                   const Section: string);
procedure WriteFormPlacement(Form: TForm);
procedure WriteFormPlacementSection(Form: TForm;
                                    const Section: string);

implementation

const
  { フォームの位置情報に対するキーワード名の定義 }
  PlaceFlags = 'PlaceFlags';
  PlaceShowCmd = 'PlaceShowCmd';
  PlaceMinPosX = 'PlaceMinPos.X';
  PlaceMinPosY = 'PlaceMinPos.Y';
  PlaceMaxPosX = 'PlaceMaxPos.X';
  PlaceMaxPosY = 'PlaceMaxPos.Y';
  PlaceNormalPosLeft = 'PlaceNormalPos.Left';
  PlaceNormalPosTop = 'PlaceNormalPos.Top';
  PlaceNormalPosRight = 'PlaceNormalPos.Right';
  PlaceNormalPosBottom = 'PlaceNormalPos.Bottom';

{ フォーム位置の読み込み }
procedure ReadFormPlacement(Form: TForm);
begin
  { フォームのクラス名をセクションとしてフォーム位置を読み込む }
  ReadFormPlacementSection(Form, Form.ClassName);
end;

{ フォーム位置の読み込み（セクション指定付） }
procedure ReadFormPlacementSection(Form: TForm;
                                   const Section: string);
var
  IniFile: TRegIniFile;
  Placement: TWindowPlacement;
begin
  IniFile := TRegIniFile.Create('Software¥DelphiApp¥'
             + ChangeFileExt(ExtractFileName(Application.ExeName), ''));
  try
    with Placement, IniFile do
    begin
      Length := SizeOf(TWindowPlacement);
      { 現在の位置情報を取得する }
      GetWindowPlacement(Form.Handle, @Placement);
      { 初期化ファイルから位置情報を読み出す }
      Flags := ReadInteger(Section, PlaceFlags, Flags);
      ShowCmd := ReadInteger(Section, PlaceShowCmd, ShowCmd);
      ptMinPosition.X := ReadInteger(Section, PlaceMinPosX,
                                     ptMinPosition.X);
```

```
      ptMinPosition.Y := ReadInteger(Section, PlaceMinPosY,
                                     ptMinPosition.Y);
      ptMaxPosition.X := ReadInteger(Section, PlaceMaxPosX,
                                     ptMaxPosition.X);
      ptMaxPosition.Y := ReadInteger(Section, PlaceMaxPosY,
                                     ptMaxPosition.Y);
      with rcNormalPosition do
      begin
        Left := ReadInteger(Section, PlaceNormalPosLeft, Left);
        Top := ReadInteger(Section, PlaceNormalPosTop, Top);
        Right := ReadInteger(Section, PlaceNormalPosRight, Right);
        Bottom := ReadInteger(Section, PlaceNormalPosBottom, Bottom);
      end;
      { 位置情報をウィンドウに反映させる }
      SetWindowPlacement(Form.Handle, @Placement);
    end;
  finally
    IniFile.Free;
  end;
end;

procedure WriteFormPlacement(Form: TForm);
begin
  WriteFormPlacementSection(Form, Form.ClassName);
end;

procedure WriteFormPlacementSection(Form: TForm;
                                    const Section: string);
var
  IniFile: TRegIniFile;
  Placement: TWindowPlacement;
begin
  IniFile := TRegIniFile.Create('Software¥DelphiApp¥'
             + ChangeFileExt(ExtractFileName(Application.ExeName), ''));
  try
    with Placement, IniFile do
    begin
      Length := SizeOf(TWindowPlacement);
      { 現在の位置情報を取得する }
      GetWindowPlacement(Form.Handle, @Placement);
      { 初期化ファイルに位置情報を書き出す }
      WriteInteger(Section, PlaceFlags, Flags);
      WriteInteger(Section, PlaceShowCmd, ShowCmd);
      WriteInteger(Section, PlaceMinPosX, ptMinPosition.X);
      WriteInteger(Section, PlaceMinPosY, ptMinPosition.Y);
      WriteInteger(Section, PlaceMaxPosX, ptMaxPosition.X);
      WriteInteger(Section, PlaceMaxPosY, ptMaxPosition.Y);
      with rcNormalPosition do
```

```
      begin
        WriteInteger(Section, PlaceNormalPosLeft, Left);
        WriteInteger(Section, PlaceNormalPosTop, Top);
        WriteInteger(Section, PlaceNormalPosRight, Right);
        WriteInteger(Section, PlaceNormalPosBottom, Bottom);
      end;
    end;
  finally
    IniFile.Free;
  end;
end;

end.
```

付録FD　CHAP3¥USEWPLC.DPR

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。

ヘルプ　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。

付録FD　CHAP3¥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)|Image Editor］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ひと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。

付録FD　CHAP3¥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［ﾌﾟﾛｼﾞｪｸﾄ(P)｜ｵﾌﾟｼｮﾝ(O)｜ｱﾌﾟﾘｹｰｼｮﾝ］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。

付録FD　CHAP3¥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;
```

付録FD　CHAP3¥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。

付録FD　CHAP3¥NOTITLE.DPR

Q. 矩形（長方形）でないフォームを作成することはできませんか。

Windows95では、ウィンドウに「リージョン」(領域)を割り当てられます。リージョンには矩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;
```

付録FD　CHAP3¥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が、終了することが決まった場合の処理はWM_ENDSESSIONメッセージを使います。

付録FD　CHAP3¥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;
```

付録FD　CHAP3¥FIXPOS.DPR

Q. **[プロジェクト(P)]-[オプション(O)]-[アプリケーション] 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。ただし、Applicationオブジェクトは目に見えないため、オブジェクトインスペクタでイベントハンドラを定義することはできません。フォームのOnCreateイベントハンドラなどで明示的にイベントハンドラを指定する必要があります。

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
      { メッセージが送られるウィンドウハンドルが MDICLIENT で }
      { 送られたメッセージが F1 キーが押されたものであれば、 }
      { フォーム自身の HelpContext を使ってヘルプを呼び出す }
      Application.HelpContext(HelpContext);
      { メッセージを処理したことを示す }
      Handled := True;
    end;
end;
```

付録FD　CHAP3¥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、簡単なWindows APIを使って実現できます。必要な処理は、フォームがファイルのドロップを受け入れるようにして、WM_DROPFILESメッセージを処理することです。

```
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
  private
    procedure WMDropFiles(var Msg: TWMDropFiles);
              message WM_DROPFILES;
  end;
  ...
procedure TForm1.FormCreate(Sender: TObject);
begin
  { フォームがファイルのドロップを受け入れるための設定 }
  DragAcceptFiles(Handle, True);
end;

procedure TForm1.WMDropFiles(var Msg: TWMDropFiles);
var
  FileName: array [0..255] of Char;
begin
  { ドロップされたファイル名の取得 }
  DragQueryFile(Msg.Drop, 0, FileName, SizeOf(FileName));
  { Memo1 コンポーネントに読み込む }
  Memo1.Lines.LoadFromFile(StrPas(FileName));
  { ドロップ処理の終了 }
  DragFinish(Msg.Drop);
end;
```

Windows 3.1では、アプリケーションがアイコン化されている場合のために、Applicationオブジェクトへのメッセージも処理する必要がありました。Windows95では、最小化されてタスクバーに表示されているアプリケーションには、直接ファイルをドロップできません。タスクバーに表示されたアプリケーション上でマウスのボタンを押したままにしていると、ファイルをドロップできるようアプリケーションの大きさが復元されます。

付録FD　CHAP3¥DRGFILE.DPR

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ておくとよいでしょう。

```
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;
```

付録FD　CHAP3¥KEEPMIN.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。
さらに、いくつかのコンポーネントを新しく作成しています。

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。

付録FD　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型はであらわされる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gdFocued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;
```

付録FD　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。

付録FD　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;
```

付録FD　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;
```

付録FD　CHAP4¥STRGRID.DPR

Q. 実行時にパネルの大きさをマウスで変更させたいのですが、どうすればよいでしょうか。

パネルの右端や下端でマウスを押したり、移動するときのイベントハンドラを定義すれば、パネルの大きさを変更できるようになります。より単純な方法として、細いパネルを配置してサイズ変更用の領域として使うことができます。

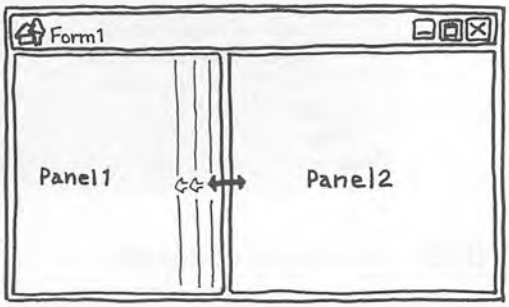
たとえば、フォームを上下に分割する場合、AlignをalTopにしたPanel1、Panel2とAlignをalClientにしたPanel3を配置します。ここでPanel2は分割線として利用するものです。Panel2のCursorはcrVSplitにし、また高さ（Height）は3～4くらいにしておきます。ここで、Panel2のOnMouseMoveイベントハンドラを次のように定義すれば、この細いバーを使ってPanel1とPanel3の分割領域を変更できます。

```
procedure TForm1.Panel2MouseMove(Sender: TObject; Shift: TShiftState;
    X, Y: Integer);
begin
  if ssLeft in Shift then
    Panel1.Height := Panel1.Height + Y;
end;
```

Panel1とPanel3の関係が左右の場合には、Panel2のAlignをalRight、CursorをcrHSplitにして、OnMouseMoveイベントハンドラを次のように定義します。

```
procedure TForm1.Panel2MouseMove(Sender: TObject; Shift: TShiftState;
    X, Y: Integer);
begin
  if ssLeft in Shift then
    Panel1.Width := Panel1.Width + X;
end;
```

付録FD　CHAP4¥RESIZE.DPR



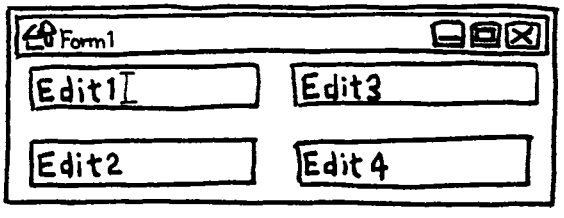
Q. Editコンポーネントをいくつか配置して、タブキーの代わりに矢印キーやリターンキーで項目を移動させようと考えています。どのようにプログラムすればよいでしょうか。

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。リターンキーは、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〕、〔リターン〕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。

重要　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コンポーネントを下のように配置した場合に、矢印キーやリターンキー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。



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、リターン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;
```

付録FD　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;
```

付録FD　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。

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）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。

付録FD　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;
```

付録FD　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);
```

付録FD　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;
```

付録FD　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、テキストの中から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。

付録FD　CHAP4¥BIGEDIT.DPR

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行入力において右寄せを使うことができません。

ごく簡単な方法として、Editコンポーネントの代わりにMemoコンポーネントを使いLinesを空（0行）にし、WantReturnsとWordWrapプロパティをFalseにしておくことができます。Alignmentプロパティを taRightJustifyにしておけば、右寄せで1行入力できます。ただし、〔Ctrl〕+〔J〕が押されたり行末で右矢印が押されると行が分割されてしまうため、限定的に利用するかキー入力イベント（OnKeyPressやOnKeyDown）でこれらのキー入力をキャンセルすることが望ましいでしょう。

付録FD　CHAP4¥CALCDEMO.DPR

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。次のプログラム(TEXTCTRL.PAS)は、このSTATICコントロールを利用してウィンドウコントロールとしてテキストを表示するコンポーネントを作成する簡単なプログラム例です。このコンポーネントを使えば、他のウィンドウコントロールに対しても前後関係を変更できます。ただし、STATICコントロールには文字以外を透明にするというオプションはありません。

付録FD　QACOMPO¥TEXTCTRL.PAS

```
unit TextCtrl;
interface
uses Windows, Messages, SysUtils, Classes, Controls, Graphics, Forms,
     Dialogs;

type
  { TWinControl からテキストコントロールを派生させる }
  TCustomText = class(TWinControl)
  private
    FAlignment: TAlignment;
    FShowAccelChar: Boolean;
    procedure SetAlignment(Value: TAlignment);
    procedure SetShowAccelChar(Value: Boolean);
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    property Alignment: TAlignment read FAlignment write SetAlignment
             default taLeftJustify;
    property ShowAccelChar: Boolean read FShowAccelChar
             write SetShowAccelChar default True;
  public
    constructor Create(AOwner: TComponent); override;
  end;

  TText  = class(TCustomText)
  published
    property Alignment;
    property Caption;
    property Color;
    property DragCursor;
    property DragMode;
    property Enabled;
    property Font;
    property ParentColor;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property ShowAccelChar;
    property Visible;
    property OnClick;
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
  end;
```

```
procedure Register;

implementation

constructor TCustomText.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := [csClickEvents, csSetCaption, csDoubleClicks];
  Width := 65;
  Height := 17;
  FShowAccelChar := True;
end;

procedure TCustomText.SetAlignment(Value: TAlignment);
begin
  if FAlignment <> Value then
  begin
    FAlignment := Value;
    RecreateWnd;
  end;
end;

procedure TCustomText.SetShowAccelChar(Value: Boolean);
begin
  if FShowAccelChar <> Value then
  begin
    FShowAccelChar := Value;
    RecreateWnd;
  end;
end;

procedure TCustomText.CreateParams(var Params: TCreateParams);
const
  Alignments: array[TAlignment] of Word = (SS_LEFT, SS_RIGHT, SS_CENTER);
  ShowAccelChars: array[Boolean] of Word = (SS_NOPREFIX, 0);
begin
  inherited CreateParams(Params);
  CreateSubClass(Params, 'STATIC');
  Params.Style := Params.Style or Alignments[Alignment]
                  or ShowAccelChars[FShowAccelChar];
end;
procedure Register;
begin
  RegisterComponents('QABook2', [TText]);
end;

end.
```

付録FD　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。

付録FD　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。

付録FD　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、ひとつのオブジェクト（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。

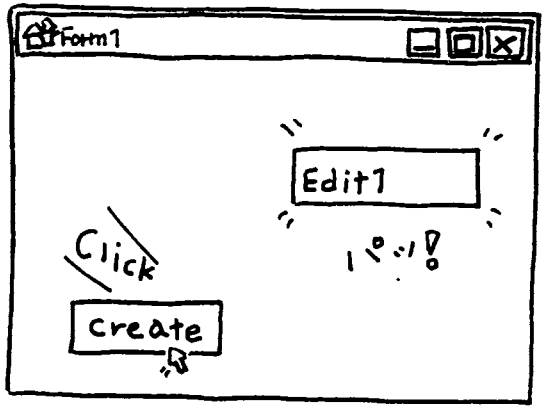
コンポーネント名をあらわすNameも重要です。設計時にフォームに配置したコンポーネントは、自動的にフォームのメンバ名と同じ名前が付けられますが、動的に生成したコンポーネントは明示的にNameプロパティを設定しなければ名前は空になってしまいます。名前が空の場合は、FindComponentなどでコンポーネントを見つけることができなくなります。

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。

付録FD　CHAP4¥DYNEDIT.DPR



Q. ヒント表示のフォントや色を変更するにはどうすればよいでしょうか。

Delphiの統合開発環境で、マウスカーソルをスピードバーやコンポーネントパレットに移動させるとボタンやアイコンの意味を示す簡単な文字列が表示されます。Delphiでは、これを（ヘルプ）ヒント表示とかフライバイヘルプ(fly-by help)と呼んでいます。また、ツールチップとかバルーンヘルプと呼ぶこともあります。作成するアプリケーションでヒント表示を利用するためには、コンポーネントのHintプロパティに文字列を設定して、ShowHintプロパティをTrueにするだけです。

ヒント表示のためのフォントには、Windows95ではウィンドウの非クライアント領域（タイトルバーなど）と同じフォントが使われます。それ以外では、「MS Pゴシック」の9ポイントフォントが使われます（日本語版の場合）。

アプリケーションプログラムで独自のフォントを指定したい場合は、THintWindowから新たなクラスを派生してHintWindowClassに代入します。ただし、HintWindowClassの内容を反映させるためにApplicationのShowHintプロパティをいったんFalseにして、再びTrueに戻す必要があります。たとえば、次のようなクラスを作成すると赤い文字でArial 16pt.フォントを使ったヒント表示が使えるようになります。

```
type
  { 新しいヒントウィンドウクラスの定義 }
  TNewHintWindow = class(THintWindow)
  public
    constructor Create(AOwner: TComponent); override;
  end;

constructor TNewHintWindow.Create(AOwner: TComponent);
begin
  inherited Create(AOnwer);
  with Canvas.Font do
  begin
    Color := clRed;     { 文字を赤で描画する }
    Name := 'Arial';    { 英文フォントを指定する }
    Size := 16;         { フォントの大きさを 16pt. にする }
  end;
end;
```

```
initialization
  { ヒントウィンドウクラスを割り当てる }
  HintWindowClass := TNewHintWindow;
  Application.ShowHint := False;
  Application.ShowHint := True;
end;
```

THintWindowというクラスには、デフォルトのヒント表示の動作が定義されています。このクラスがHintWindowClassというグローバル変数に代入されています。C++などに慣れていると、クラスそのものを変数に代入するということに違和感があるかもしれませんが、Object Pascalではオブジェクト（インスタンス）だけでなくクラス型というもので、クラスそのものを表わす変数が使えます。これを使えば、THintWindow.Create(Self)のようにする代わりに、HintWindowClass.Create(Self)とできます。

ヒント表示の方法を変更するために、Applicationオブジェクトのプロパティやイベントを使う方法があります。Applicationには、ヒント表示のためにHintColor、HintPause、HintShortPause、HintHidePauseといったプロパティやOnShowHintというイベントがあります。Applicationは、フォームやフォーム上のコンポーネントと違ってまったく見えないオブジェクトとして存在するため、これらは実行時に設定します。

HintColorは、ヒント文字列の背景色を指定します。デフォルトでは$ 80FFFF（明るい黄色）という数値が割り当てられていますが、clAquaなどの色シンボルや他のRGB値を指定できます。なお、ヒント文字列の背景色は、この設定が優先されるため上記のようなHintWindowの派生クラスでColorプロパティを設定しても無意味です。

HintPauseは、マウスカーソルがコントロール上にきてから実際にヒント文字列を表示するまでの時間をミリ秒(ms)で指定するものです。デフォルトでは800msですが、もっと早めたい場合は数値を小さくします。HintShortPauseは、ヒントを表示している状態から別のコンポーネントのヒントを表示する際にかかる時間です。HintHidePauseは、表示しているヒントを閉じるまでの時間です。

OnShowHintは、ヒント文字列を表示する前に表示内容や情報が渡されるもので、これを変更することでヒント文字列の表示位置や最大幅、背景色を変更できます。OnShowHintのイベントハンドラは、procedure ShowHintHandler(var HintStr: string; var CanShow: Boolean; var HintInfo: THintInfo);と

いう形式のメソッドになります。

これらをメインフォームのOnCreateイベントハンドラなどで設定します。

```
type
  TForm1 = class(TForm)
    ...
    procedure FormCreate(Sender: TObject);
  private
    procedure ShowHint(var HintStr: string; var CanShow: Boolean;
                       var HintInfo: THintInfo);
    ...
  end;

{ メインフォームの OnCreate イベントハンドラで }
{ アプリケーションのヒント情報を設定する }
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.HintPause := 300;
  Application.HintColor := clAqua;
  Application.OnShowHint := ShowHint;
end;

{ Application の OnShowHint イベントハンドラ用のメソッド }
procedure TForm1.ShowHint(var HintStr: string; var CanShow: Boolean;
                          var HintInfo: THintInfo);
begin
  with HintInfo do
  begin
    { ヒント表示の最大幅の設定 }
    HintMaxWidth := 100;
    { コントロールが TButton であれば背景色を水色にする }
    if HintControl is TButton then
      HintColor := clAqua;
  end;
end;
```

前述のヒントウィンドウでは、フォントの種類や背景色だけを変更していますが、THintWindowクラスを大幅に書換えれば、ヒント表示の形式も大きく変更できます。次のヒントウィンドウクラスは、枠がなく背景を透明にしてヒント文字列を表示します。ここでは、ヒント文字列の描画そのものも自分で処理しています。これを拡張することで、ヒント表示の枠に飾りをつけたり、より凝ったものにすることができます。

```
type
  { 新しいヒントウィンドウクラス }
  TNewHintWindow = class(THintWindow)
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    procedure Paint; override;
  public
    constructor Create(AOwner: TComponent); override;
    procedure ActivateHint(Rect: TRect; const AHint: string); override;
    procedure WMEraseBkgnd(var Msg: TMessage); message WM_ERASEBKGND;
  end;

{ TNewHintWindow のコンストラクタ }
constructor TNewHintWindow.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Canvas.Font プロパティを設定する }
end;

{ 背景を塗りつぶすための WM_ERASEBKGND メッセージハンドラ }
procedure TNewHintWindow.WMEraseBkgnd(var Msg: TMessage);
begin
  { 背景を描画し直さずに終了する }
  Msg.Result := 1;
end;

{ ウィンドウを作成するときのパラメータを設定する }
procedure TNewHintWindow.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  Params.Style := WS_POPUP or WS_DISABLED;  { 枠 (WS_BORDER) を指定しない }
end;

{ 自分自身でヒント文字列を描画する }
procedure TNewHintWindow.Paint;
var
  R: TRect;
begin
  R := ClientRect;
  Inc(R.Left, 1);
  OffsetRect(R, 0, 10);  { 描画位置を少し下にずらす }
  DrawText(Canvas.Handle, PChar(Caption), -1, R,
    DT_LEFT or DT_NOPREFIX or DT_WORDBREAK);
end;
{ 描画する領域を下に広げる }
procedure TNewHintWindow.ActivateHint(Rect: TRect; const AHint: string);
begin
  Inc(Rect.Bottom, 10);
  inherited ActivateHint(Rect, AHint);
end;
```

付録FD　CHAP4¥HINTWIN.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。

付録FD　CHAP4¥DRAWIMG.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。

重要 Delphi 2.0の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ません。

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FD　CHAP4¥APPEXCEP.DPR

Q. 新しいコンポーネントを作成するときに、文字列リストを使いたいのでTStrings型のプロパティを作ったのですが正しく動作しません。TStrings型のプロパティを使うときの注意点を教えてください。

TStringsは、classを使って定義されているクラスです。このため、新しいコンポーネントのコンストラクタでTStringListオブジェクトを作成し、デストラクタで解放する必要があります。

TStringsではなくTStringListオブジェクトを使うのは、TStringsには文字列を保持しておく仕組みが提供されていないためです。TStringListオブジェクトを使えば、自動的にメモリが確保され文字列が保持されます。もし、Memoコンポーネントのように Windows の EDIT コントロール自身を文字列の保持領域として使う場合は、独自の入出力インターフェースを定義する必要があります。Memoコンポーネントに関するソースコードは、¥DELPHI¥DOCにあるSTDCTRLS.PASに含まれています。

また、コンポーネントのコンストラクタで文字列リストが変更された場合のイベントハンドラを定義する必要があります。そうしないと、文字列リストの変更がコンポーネント自身に反映されなくなります。

さらに、TStrings型のプロパティは必ず設定メソッドを定義して、単純代入ではなくAssignを使って文字列を割り当てるようにしなければなりません。TStrings型はclassで定義されるクラス型なので、変数の代入はオブジェクトを指すポインタどうしの代入にすぎないためです。

以下に、描画ボックス(PaintBox)を拡張して、文字列リストの内容を描画する新しいコンポーネントのプログラム例を示します。

付録FD　QACOMPO¥PBOXLIST.PAS

```
{ 文字列リストを描画する描画ボックス }
unit Pboxlst;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls;

type
  TPaintBoxList = class(TPaintBox)
  private
    FItems: TStrings;
    procedure SetItems(Value: TStrings);
    procedure ItemChanging(Sender: TObject);
  protected
    procedure Paint; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Items: TStrings read FItems write SetItems;
  end;

procedure Register;

implementation

{ コンストラクタ }
constructor TPaintBoxList.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { TStringList オブジェクトの作成 }
  FItems := TStringList.Create;
  { 文字列が変更されたときのイベントハンドラを指定 }
  TStringList(FItems).OnChanging := ItemChanging;
end;

{ デストラクタ }
destructor TPaintBoxList.Destroy;
begin
  { TStringList オブジェクトを解放 }
  FItems.Free;
  inherited Destroy;
end;
```

```
{ Items プロパティの設定メソッド }
procedure TPaintBoxList.SetItems(Value: TStrings);
begin
  { Assign メソッドを使って文字列リストを割り当てる }
  FItems.Assign(Value);
end;

{ Items プロパティの文字列が変更されたときのイベントハンドラ }
procedure TPaintBoxList.ItemChanging(Sender: TObject);
begin
  Invalidate;
end;

{ 描画ボックス内に文字列リストを描画する }
procedure TPaintBoxList.Paint;
var
  i: Integer;
  TempHeight: Longint;
  TempCount: Integer;
begin
  if FItems.Count = 0 then
    Exit;
  Canvas.Brush.Style := bsClear;
  Canvas.Font := Font;
  TempHeight := Height - Canvas.TextHeight('H');
  if TempHeight < 0 then
    TempHeight := 0;
  TempCount := FItems.Count - 1;
  if TempCount = 0 then
    TempCount := 1;
  for i := 0 to TempCount do
    Canvas.TextOut(0, TempHeight * i div TempCount, FItems[i]);
end;

{ コンポーネントの登録 }
procedure Register;
begin
  RegisterComponents('QABook2', [TPaintBoxList]);
end;

end.
```

付録FD　CHAP4¥USEPBLST.DPR

Q. Memoコンポーネントに、ファイルマネージャからファイルをドロップさせたいのですがどうすればよいでしょうか。

フォームをドロップ可能にする例が、第3章「アプリケーション／フォーム」(93ページ) に書かれています。さらに、ファイルマネージャからのドロップを受け入れるようMemoコンポーネントを拡張できます。

付録FD　QACOMPO¥DRAGMEMO.PAS

```
unit Dragmemo;
{
  このコンポーネントは、ファイルマネージャからのファイルのドロップを
  受け入れるように、TMemo を継承したものです。ファイルマネージャから
  ファイルがドロップされると、OnDropFiles イベントが発生し、ドロップ
  された場所と、その場所がコンポーネントのクライアント領域かどうかを
  イベントハンドラに渡します。
 OnDropFiles イベントハンドラは、次のような形式です。

   procedure (Sender: TObject; X, Y: Integer; InClient: Boolean);

  ドロップされたファイル名は、DropFiles プロパティで参照できます。
  これは TStrings 型のプロパティで、KeepFiles プロパティが True のと
  き（デフォルト）、ドロップされたファイル名リストは次にドロップされ
  るまで保持されます。KeepFiles が False のとき、OnDropFiles イベン
  トハンドラ以外では DropFiles は空になっています。
}
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ShellAPI;

type
  TDropFilesEvent = procedure(Sender: TObject; X, Y: Integer;
                              InClient: Boolean) of object;
  TDragMemo = class(TMemo)
  private
    FDropFiles: TStrings;
    FKeepFiles: Boolean;
    FOnDropFiles: TDropFilesEvent;
    procedure WMDropFiles(var Msg: TWMDropFiles);
              message WM_DROPFILES;
  protected
    procedure CreateWnd; override;
```

```
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    property DropFiles: TStrings read FDropFiles;
  published
    property KeepFiles: Boolean read FKeepFiles write FKeepFiles
            default True;
    property OnDropFiles: TDropFilesEvent
            read FOnDropFiles write FOnDropFiles;
  end;

procedure Register;

implementation

{ TDragMemo コンポーネント }
constructor TDragMemo.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { ドロップ終了後もファイルリストを保持する }
  FKeepFiles := True;
  { DropFiles プロパティのために文字列リストを作成する }
  FDropFiles := TStringList.Create;
end;

destructor TDragMemo.Destroy;
begin
  FDropFiles.Free;
  inherited Destroy;
end;

procedure TDragMemo.CreateWnd;
begin
  inherited CreateWnd;
  { ウィンドウが作成されるときに、ファイルのドロップを受け入れる }
  DragAcceptFiles(Handle, True);
end;

{ ファイルがドロップされたときのメッセージ処理 }
procedure TDragMemo.WMDropFiles(var Msg: TWMDropFiles);
var
  I, Num: Word;
  Pt: TPoint;
  InClient: Boolean;
  Temp: array [0..255] of Char;
begin
  { FOnDropFiles イベントハンドラが定義されているときに処理 }
  Num := DragQueryFile(Msg.Drop, $FFFFFFFF, nil, 0);
  FDropFiles.Clear;
```

```
  { イベントハンドラが割り当てられているか KeepFiles プロパティが }
  { True のときに、ファイル名リストを作成 }
  if Assigned(FOnDropFiles) or FKeepFiles then
    for I := 0 to Num - 1 do
    begin
      DragQueryFile(Msg.Drop, I, Temp, SizeOf(Temp));
      FDropFiles.Add(StrPas(Temp));
    end;
  { イベントハンドラが割り当てられているときの処理 }
  if Assigned(FOnDropFiles) then
  begin
    InClient := DragQueryPoint(Msg.Drop, Pt);
    FOnDropFiles(Self, Pt.X, Pt.Y, InClient);
  end;
  if not FKeepFiles then
    FDropFiles.Clear;
  { ドラッグのためのメモリを解放 }
  DragFinish(Msg.Drop);
end;

procedure Register;
begin
  RegisterComponents('QABook2', [TDragMemo]);
end;

end.
```

DragMemoコンポーネントを配置して、OnDropFilesイベントハンドラを次のように定義すれば、ドロップされたファイルのリストが編集領域に表示されます。

```
procedure TForm1.DragMemo1DropFiles(Sender: TObject; X, Y: Integer;
            InClient: Boolean);
begin
  Memo1.Lines.Assign(Memo1.DropFiles);
end;
```

付録FD　CHAP4¥USEDRGM.DPR

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。以下に、他のウィンドウコントロールとともに使うためのコンポーネントのプログラムを示します。

付録FD　QACOMPO¥COMPOSTR.PAS

```
unit CompoStr;
{
        TCompoString:

         TCompoString は、かな漢字変換を使って漢字を入力したときに
        使われた構成文字列（読み仮名）を取得するコンポーネントです。
         Control プロパティには、読み仮名を取得する対象となる
        コントロールを指定します。ここには、Edit、Memo、MaskEdit、
        RichEdit、DBEdit などの文字列を入力するためのコントロールを
        指定できます。
          対象となるコントロールで入力した文字が確定すると、
        OnCompositionStr イベントが発生します。このイベントハンドラの
        Value（string 型）引数に読み仮名が渡されます。

        ※注意 ComboBox はコンボボックス自身と編集ウィンドウ自身の
            コントロールハンドルが違うため指定しても動作しません。
}
interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, Imm;

type
  TCompositionStrEvent = procedure (Sender: TObject; Value: string) of
object;

  TCompoString = class(TComponent)
  private
    FControl: TWinControl;
    FOnCompositionStr: TCompositionStrEvent;
    FControlInstance: TFarProc;
    FDefControlProc: TFarProc;
    procedure SetControl(Value: TWinControl);
    procedure ControlWndProc(var Message: TMessage);
```

```
  protected
    procedure Notification(AComponent: TComponent;
              Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Control: TWinControl read FControl write SetControl;
    property OnCompositionStr: TCompositionStrEvent
             read FOnCompositionStr write FOnCompositionStr;
  end;

procedure Register;

implementation

procedure TCompoString.SetControl(Value: TWinControl);
begin
  if FControl <> Value then
  begin
    if not (csDesigning in ComponentState) and (FControl <> nil)
       and FControl.HandleAllocated then
      SetWindowLong(FControl.Handle, GWL_WNDPROC,
                    Longint(FDefControlProc));
    FControl := Value;
    if not (csDesigning in ComponentState) and (FControl <> nil) then
    begin
      FDefControlProc := TFarProc(GetWindowLong(FControl.Handle,
                                  GWL_WNDPROC));
      SetWindowLong(FControl.Handle, GWL_WNDPROC,
                    Longint(FControlInstance));
    end;
  end;
end;

procedure TCompoString.ControlWndProc(var Message: TMessage);
var
  Imc: HIMC;
  Len: Longint;
  Str: string;
begin
  with Message do
  begin
    if Assigned(FOnCompositionStr) and (Msg = WM_IME_COMPOSITION)
      and ((LParam and GCS_RESULTREADSTR) <> 0) then
    begin
      Imc := ImmGetContext(FControl.Handle);
      Len := ImmGetCompositionString(Imc, GCS_RESULTREADSTR, nil, 0);
      SetLength(Str, Len + 1);
```

```
      ImmGetCompositionString(Imc, GCS_RESULTREADSTR, PChar(Str),
                              Len + 1);
      ImmReleaseContext(FControl.Handle, Imc);
      SetLength(Str, Len);
      FOnCompositionStr(Self, Str);
    end;
    Result := CallWindowProc(FDefControlProc, FControl.Handle, Msg,
              wParam, lParam);
  end;
end;

constructor TCompoString.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FControlInstance := MakeObjectInstance(ControlWndProc);
end;

procedure TCompoString.Notification(AComponent: TComponent;
          Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (AComponent = FControl) and (Operation = opRemove)then
    Control := nil;
end;

destructor TCompoString.Destroy;
begin
  if not (csDesigning in ComponentState) and (FControl <> nil)
     and FControl.HandleAllocated then
    SetWindowLong(FControl.Handle, GWL_WNDPROC,
                  Longint(FDefControlProc));
  FreeObjectInstance(FControlInstance);
  inherited Destroy;
end;

procedure Register;
begin
  RegisterComponents('QABook2', [TCompoString]);
end;

end.
```

このコンポーネントは、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;
```

付録FD　CHAP4¥USECOMPO.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。

付録FD　CHAP4¥ACTPAGE.DPR

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。
「Borland Database Engine の初期化中にｴﾗｰが発生しました(ｴﾗｰ $2109)」

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2570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OnDrawDataCell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 DrawDataCell(Sender: TObject; const Rect: TRect; Field: TField;
    State: TGridDrawState);
```

また、DBGridには内部でDefaultDrawDataCellという非公開の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、OnDrawDataCellイベントに次のイベントハンドラを定義すればよいのです（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.DBGrid1DrawDataCell(Sender: TObject; const Rect: TRect;
  Field: TField; State: TGridDrawState);
begin
  DBGrid1.DefaultDrawDataCell(Rect, Field, 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を変更しておけばよいのです。たとえば、フォーカスのあるセルを背景を水色、文字を青で描画するためには次のようにします。

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect: TRect;
  Field: TField; State: TGridDrawState);
begin
  if gdFocused in State then
  begin
    DBGrid1.Canvas.Brush.Color := clAqua;
    DBGrid1.Canvas.Font.Color := clBlue;
  end;
  DBGrid1.DefaultDrawDataCell(Rect, Field, State);
end;
```

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「データセル」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。

付録FD　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ひと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。

ヘルプ　ローカルテーブルに対して利用できるSQL文については、オンラインヘルプの「ローカルSQL」で表示されるヘルプ画面で、トピック「ローカルSQLの使い方」を選んでください。

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。

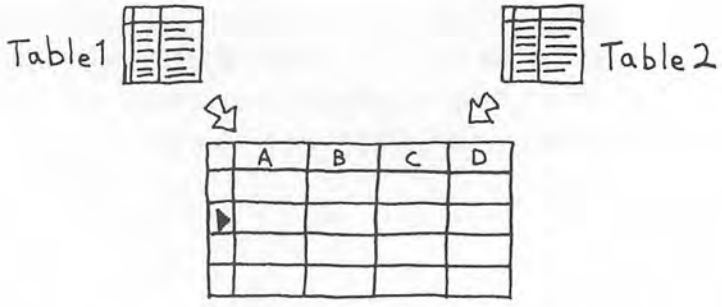
```
procedure TForm1.Table1CompanyGetText(Sender: TField; var Text: string;
  DisplayText: Boolean);
begin
  if Table2.FindKey([Table1CustNo.Value]) then
    Text := '[' + Table2.FieldByName('Company').AsString + ']';
end;
```

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。

重要　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。

付録FD　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。これは、TBookmarkList 型のプロパティで、表1に示すメソッドやプロパティがあります。

表1　TBookmarkList 型

メソッド／プロパティ	解説
Clear メソッド	選択したレコード情報の解放
Delete メソッド	選択したレコードの削除
Find メソッド	ブックマークが選択状態かどうか調べる
IndexOf メソッド	選択リスト中のブックマークの位置
Refresh メソッド	選択リストの更新
Count プロパティ	選択したレコードの数
CurrentRowSelected プロパティ	カレントレコードが選択状態かどうか
Items プロパティ	ブックマークで示された選択リスト

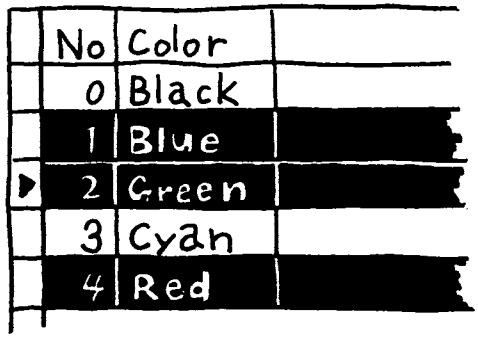
複数のレコード選択を試すため、次の手順でフォームを作成します。

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。

付録FD　CHAP5¥DBGMREC.DPR



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。

参照　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「コンポーネント」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;
```

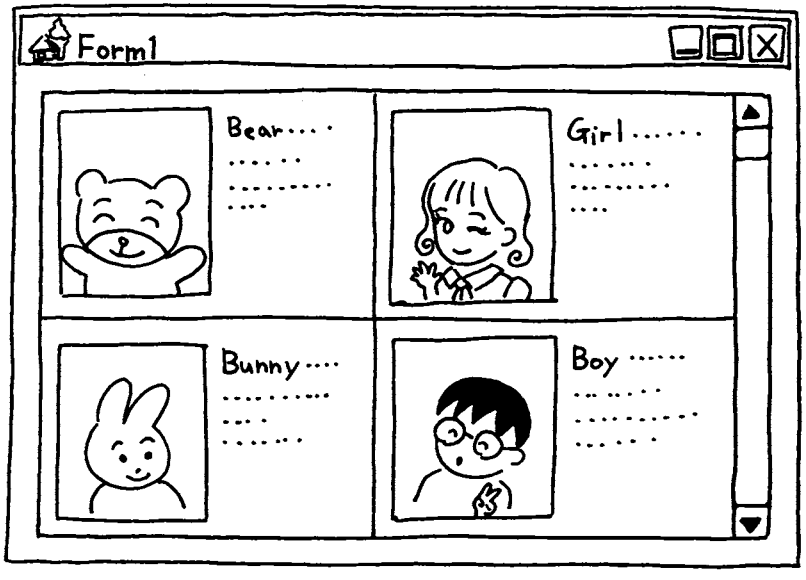
付録FD　CHAP5¥DYNTABLE.DPR

Q. DBCtrlGridにDBImageを配置したいのですが、「指定されたｺﾝﾄﾛｰﾙはDBCtrlGrid内では使えません」というエラーメッセージが表示されてしまいます。

Delphi Developer 2.0以上で提供されているDBCtrlGridは、データコントロールを任意に配置して複数のレコードを繰返し表示するためのものです。ここには、DBTextやDBEditのように単純型の項目を表示するためのコントロールは配置できますが、DBMemoやDBImageのようにBLObに対応するコントロールは配置できません。これは、BLOb情報を読み込む速度が、SQLサーバー上のデータでは非常に遅くなることがあるためです。

どうしても、DBMemoやDBImageを配置したい場合、次のようにコンポーネントを拡張してcsReplicatableオプションを有効にします。この拡張されたコンポーネントは、DBCtrlGridに配置できます。

付録FD　QACOMPO¥DBBLOBEX.PAS



```
unit DBBlobEx;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, DBCtrls;

type
  TDBMemoEx = class(TDBMemo)
  public
    constructor Create(AOwner: TComponent); override;
  end;

  TDBImageEx = class(TDBImage)
  public
    constructor Create(AOwner: TComponent); override;
  end;

procedure Register;

implementation

constructor TDBMemoEx.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { csReplicatable オプションをデフォルトスタイルに追加する }
  ControlStyle := ControlStyle + [csReplicatable];
end;

constructor TDBImageEx.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { csReplicatable オプションをデフォルトスタイルに追加する }
  ControlStyle := ControlStyle + [csReplicatable];
end;

procedure Register;
begin
  RegisterComponents('QABook2', [TDBMemoEx, TDBImageEx]);
end;

end.
```

付録FD　CHAP5¥USEDBEX.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。

テーブルを作成する必要がある場合、ParadoxやdBASEなどのデータベースアプリケーションを使えます。また、Delphi Developer 2.0やDelphi Client/Server Suite 2.0では、16ビット版Delphi and Database Toolsの付属ツールとしてDatabase Desktopが含まれていますので、これを使うこともできます（ただし、Windows95の長いファイル名などには対応していません）。あるいは、SQLサーバーであればサーバーに付属するツールや機能を使うこともできるでしょう。

こうした周辺ツールを使わずに、Delphiのプログラムだけでテーブルを作成するには、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コンポーネントで作成できない書式付きメモ型や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。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

付録FD　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;
```

付録FD　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 batAppend 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。

付録FD　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
```

付録FD　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。

付録FD　CHAP5¥TBLRANGE.DPR

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、**インデックス項目での検索；FINDINDEXFIELD** にある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。

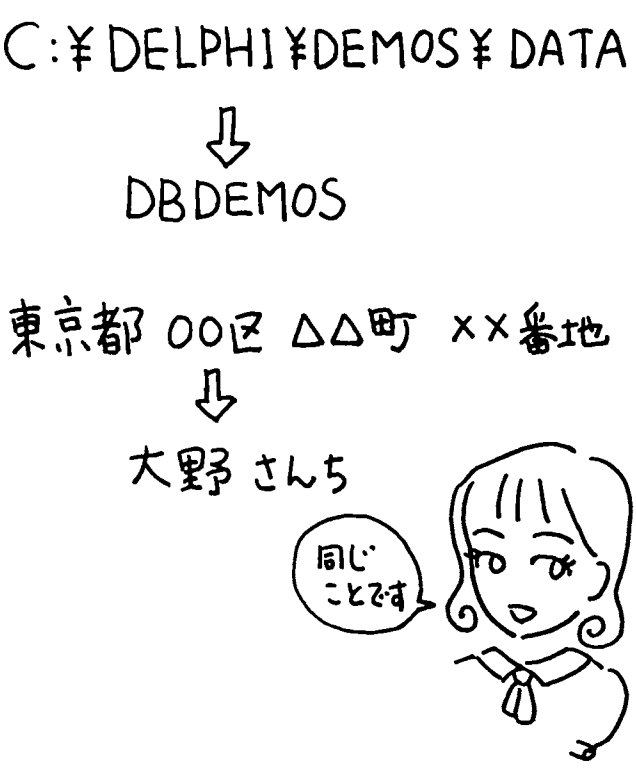
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.SetRange(['Delphi'], ['Delphi']);
end;
```

付録FD　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。

付録FD　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るので、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;
```

重要 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。

付録FD 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。

付録FD　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。

前項目に注意し、Buttonコンポーネントを拡張すれば、レコード操作のためのボタンを作ることはさほど難しくはありません。しかし、割り当てられたDataSourceがあらわすデータセットの先頭や末尾に移動したときに、自動的にボタンを無効化したり、DataSourceの変更に対応するには多少面倒な処理が必要です。

こうした処理に配慮したレコード操作ボタンのためのコンポーネントを以下に示します。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。ボタンの機能には、DBNavigatorと同じく10種類（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）あります。

付録FD　QACOMPO¥DBNAVBTN.PAS

```
unit DBNavBtn;
{
        TDBNavButton:

          レコード操作用の個別ボタンです。フォームに配置するたびに、
        [最初のﾚｺｰﾄﾞ(&F)]、[前のﾚｺｰﾄﾞ(&P)]、[次のﾚｺｰﾄﾞ(&N)]、
        [最後のﾚｺｰﾄﾞ(&L)]、[ﾚｺｰﾄﾞの挿入(&I)]、[ﾚｺｰﾄﾞの削除(&D)]、
        [ﾚｺｰﾄﾞの編集(&E)]、[ﾚｺｰﾄﾞの登録(&P)]、[編集の取り消し(&C)]、
        [ﾃﾞｰﾀ更新(&R)] ボタンが順に配置されます。
          操作対象は、DataSource プロパティで指定します。
}

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, DB, DBCtrls, DBTables;

type
  TDBNavButton = class(TButton)
  private
    FDataLink: TDataLink;
    FDataSource: TDataSource;
    FNavType: TNavigateBtn;
    procedure SetDataSource(Value: TDataSource);
    function GetDataSource: TDataSource;
    procedure SetNavType(Value: TNavigateBtn);
```

```
  protected
    procedure Click; override;
    procedure EditingChanged;
    procedure DataSetChanged;
    procedure ActiveChanged;
    procedure Notification(AComponent: TComponent;
              Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property DataSource: TDataSource read GetDataSource
             write SetDataSource;
    property Enabled default False;
    property NavType: TNavigateBtn read FNavType write SetNavType;
  end;

  TNavBtnDataLink = class(TDataLink)
  private
    FDBNavButton: TDBNavButton;
  protected
    procedure EditingChanged; override;
    procedure DataSetChanged; override;
    procedure ActiveChanged; override;
  public
    constructor Create(ANav: TDBNavButton);
    destructor Destroy; override;
  end;

procedure Register;

implementation

constructor TDBNavButton.Create(AOwner: TComponent);
var
  I: Integer;
  Btn: TNavigateBtn;
  Btns: TButtonSet;
begin
  inherited Create(AOwner);
  Enabled := False;
  Width := 100;
  with AOwner do
  begin
    Btns := [];
    for I := 0 to ComponentCount - 1 do
      if (Components[I] is TDBNavButton)and(Components[I] <> Self) then
        Btns := Btns + [TDBNavButton(Components[I]).NavType];
    FNavType := nbFirst;
    for Btn := nbFirst to nbRefresh do
```

```
      if not (Btn in Btns) then
      begin
        if Btn = nbFirst then FNavType := nbRefresh;
        NavType := Btn;
        Break;
      end;
  end;
  FDataLink := TNavBtnDataLink.Create(Self);
end;

destructor TDBNavButton.Destroy;
begin
  FDataLink.Free;
  FDataLink := nil;
  inherited Destroy;
end;

procedure TDBNavButton.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  if not (csLoading in ComponentState) then
    ActiveChanged;
  if Value <> nil then Value.FreeNotification(Self);
end;

function TDBNavButton.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBNavButton.SetNavType(Value: TNavigateBtn);
const
  NavCaptions: array [nbFirst..nbRefresh] of PChar =
    ('最初のﾚｺｰﾄﾞ(&F)', '前のﾚｺｰﾄﾞ(&P)', '次のﾚｺｰﾄﾞ(&N)',
     '最後のﾚｺｰﾄﾞ(&L)', 'ﾚｺｰﾄﾞの挿入(&I)', 'ﾚｺｰﾄﾞの削除(&D)',
     'ﾚｺｰﾄﾞの編集(&E)', 'ﾚｺｰﾄﾞの登録(&P)', '編集の取り消し(&C)',
     'ﾃﾞｰﾀ更新(&R)');
begin
  if FNavType <> Value then
  begin
    FNavType := Value;
    Caption := StrPas(NavCaptions[FNavType]);
  end;
end;

procedure TDBNavButton.Click;
begin
  inherited Click;
  with DataSource.DataSet do
    case NavType of
```

```
    nbFirst: First;
    nbPrior: Prior;
    nbNext: Next;
    nbLast: Last;
    nbInsert: Insert;
    nbDelete: Delete; { no confirmation }
    nbEdit: Edit;
    nbPost: Post;
    nbCancel: Cancel;
    nbRefresh: Refresh;
    end;
end;

procedure TDBNavButton.EditingChanged;
var
  CanModify: Boolean;
begin
  CanModify := FDataLink.Active and FDataLink.DataSet.CanModify;
  case FNavType of
  nbInsert: Enabled := CanModify;
  nbEdit: Enabled := CanModify and not FDataLink.Editing;
  nbPost, nbCancel: Enabled := CanModify and FDataLink.Editing;
  nbRefresh: Enabled := not (FDataLink.DataSet is TQuery);
  end;
end;

procedure TDBNavButton.DataSetChanged;
begin
  case NavType of
  nbFirst, nbPrior:
    Enabled := FDataLink.Active and not FDataLink.DataSet.BOF;
  nbNext, nbLast:
    Enabled := FDataLink.Active and not FDataLink.DataSet.EOF;
  nbDelete:
    Enabled := FDataLink.Active and FDataLink.DataSet.CanModify
            and (FDataLink.DataSet.BOF and FDataLink.DataSet.EOF);
  end;
end;

procedure TDBNavButton.ActiveChanged;
var
  Btn: TNavigateBtn;
begin
  if FDataLink.Active then
  begin
    DataSetChanged;
    EditingChanged;
  end
  else
    Enabled := False
```

```
end;

procedure TDBNavButton.Notification(AComponent: TComponent;
          Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (FDataLink <> nil)
     and (AComponent = DataSource) then
    DataSource := nil;
end;

constructor TNavBtnDataLink.Create(ANav: TDBNavButton);
begin
  inherited Create;
  FDBNavButton := ANav;
end;

destructor TNavBtnDataLink.Destroy;
begin
  FDBNavButton := nil;
  inherited Destroy;
end;

procedure TNavBtnDataLink.EditingChanged;
begin
  if FDBNavButton <> nil then
    FDBNavButton.EditingChanged;
end;

procedure TNavBtnDataLink.DataSetChanged;
begin
  if FDBNavButton <> nil then
    FDBNavButton.DataSetChanged;
end;

procedure TNavBtnDataLink.ActiveChanged;
begin
  if FDBNavButton <> nil then
    FDBNavButton.ActiveChanged;
end;

procedure Register;
begin
  RegisterComponents('QABook2', [TDBNavButton]);
end;

end.
```

付録FD　CHAP5¥USENAVB.DPR

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。

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、フォーム（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 イベントハンドラ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。

付録FD　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;
```

付録FD　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。

付録FD　CHAP6¥AREDRAW.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矩形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。

また、コンポーネントとして実現するプログラム例を以下に示します。このコンポーネントは矩形の対角線方向にラインを表示するものです。

付録FD　QACOMPO¥LINE.PAS

```
unit Line;
{
   このコンポーネントは、フォーム上に直線を描画するためのコンポーネントです。デフォルトでは、指定された矩形の左上から右下に線を描画します。
   直線の方向を変更するためには、StartPos プロパティを変更します。このプロパティを使って、開始点を左上(lsLeftTop)、左下(lsLeftBottom)、右下(lsRightBottom)、右上(lsRightTop)に変更できます。ただし、フォーム上でコンポーネントの位置や大きさを変更する場合は、左上(Left, Top)を起点とした大きさ(Width, Height)で表現されるため、Visual Basic のように開始点、終了点を任意の位置に設定することはできません。
   これをシミュレートするために、X1, Y1, X2, Y2 というプロパティを用意しています。これらのプロパティを変更すると、起点(Left, Top)や大きさ(Width, Height)、開始位置(StartPos)が自動的に設定されます。
   描画する線の色やスタイルは Pen プロパティを使って変更します。

   ※注意　線は自分自身の領域の中にしか描画しません。したがって、横線、縦線のように幅の狭い(または幅がない)領域には線を描画できない場合があります。これは、自分自身の領域ではなくフォームに描画するか、縦線、横線用のスタイルを追加することが考えられます。
}

interface

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

type
  TLineStart = (lsLeftTop, lsLeftBottom, lsRightBottom, lsRightTop);

  TLine = class(TGraphicControl)
```

```
  private
    FPen: TPen;
    FStartPos: TLineStart;
    FX1, FY1, FX2, FY2: Integer;
    procedure SetPen(Value: TPen);
    procedure SetStartPos(Value: TLineStart);
    procedure SetX1(Value: Integer);
    procedure SetY1(Value: Integer);
    procedure SetX2(Value: Integer);
    procedure SetY2(Value: Integer);
    procedure AdjustStartPos;
  protected
    procedure Paint; override;
    procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer);override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    procedure StyleChanged(Sender: TObject);
    property Direction: TLineStart read FStartPos write SetStartPos;
    property DragCursor;
    property DragMode;
    property Enabled;
    property ParentShowHint;
    property Pen: TPen read FPen write SetPen;
    property ShowHint;
    property StartPos: TLineStart read FStartPos write SetStartPos;
    property Visible;
    property X1: Integer read FX1 write SetX1;
    property X2: Integer read FX2 write SetX2;
    property Y1: Integer read FY1 write SetY1;
    property Y2: Integer read FY2 write SetY2;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
  end;

procedure Register;

implementation

{ TLine コンポーネントのコンストラクタ }
constructor TLine.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { デフォルトの大きさなどを設定する }
  Width := 50;
```

```
  Height := 50;
  FStartPos := lsLeftTop;
  FPen := TPen.Create;
  FPen.OnChange := StyleChanged;
end;

{ TLine コンポーネントのデストラクタ }
destructor TLine.Destroy;
begin
  FPen.Free;
  inherited Destroy;
end;

procedure TLine.Paint;
var
  XA, YA, XB, YB: Integer;
begin
  XA := Pen.Width div 2;
  XB := Width - Pen.Width + 1;
  if (StartPos = lsLeftTop) or (StartPos = lsRightBottom) then
  begin
    YA := XA;
    YB := Height - Pen.Width + 1;
  end
  else
  begin
    YA := Height - Pen.Width + 1;
    YB := XA;
  end;
  with Canvas do
  begin
    Pen := FPen;
    MoveTo(XA, YA);
    LineTo(XB, YB);
  end;
end;

procedure TLine.StyleChanged(Sender: TObject);
begin
  Invalidate;
end;

procedure TLine.SetStartPos(Value: TLineStart);
begin
  if FStartPos <> Value then
  begin
    FStartPos := Value;
    Invalidate;
  end;
end;
```

```
procedure TLine.SetPen(Value: TPen);
begin
  FPen.Assign(Value);
end;

procedure ISwap(var A, B: Integer);
var
  Tmp: Integer;
begin
  Tmp := A; A := B; B := Tmp;
end;

procedure TLine.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited SetBounds(ALeft, ATop, AWidth, AHeight);
  if StartPos in [lsLeftTop, lsLeftBottom] then
  begin
    X1 := ALeft;
    X2 := ALeft + AWidth;
  end
  else
  begin
    X1 := ALeft + AWidth;
    X2 := ALeft;
  end;
  if StartPos in [lsLeftTop, lsRightTop] then
  begin
    Y1 := ATop;
    Y2 := ATop + AHeight;
  end
  else
  begin
    Y1 := ATop + AHeight;
    Y2 := ATop;
  end;
end;

procedure TLine.AdjustStartPos;
var
  ALeft, ATop, AWidth, AHeight: Integer;
begin
  if (FX1 <= FX2) and (FY1 <= FY2) then
    FStartPos := lsLeftTop
  else if (FX1 <= FX2) and (FY1 > FY2) then
    FStartPos := lsLeftBottom
  else if (FX1 > FX2) and (FY1 > FY2) then
    FStartPos := lsRightBottom
  else
    FStartPos := lsRightTop;
  if FX1 <= FX2 then
```

```
  begin
    ALeft := FX1;
    AWidth := FX2 - FX1;
  end
  else
  begin
    ALeft := FX2;
    AWidth := FX1 - FX2;
  end;
  if FY1 <= FY2 then
  begin
    ATop := FY1;
    AHeight := FY2 - FY1;
  end
  else
  begin
    ATop := FY2;
    AHeight := FY1 - FY2;
  end;
  SetBounds(ALeft, ATop, AWidth, AHeight);
end;

procedure TLine.SetX1(Value: Integer);
begin
  if FX1 <> Value then
  begin
    FX1 := Value;
    AdjustStartPos;
  end;
end;

procedure TLine.SetY1(Value: Integer);
begin
  if FY1 <> Value then
  begin
    FY1 := Value;
    AdjustStartPos;
  end;
end;

procedure TLine.SetX2(Value: Integer);
begin
  if FX2 <> Value then
  begin
    FX2 := Value;
    AdjustStartPos;
  end;
end;
```

```
procedure TLine.SetY2(Value: Integer);
begin
  if FY2 <> Value then
  begin
    FY2 := Value;
    AdjustStartPos;
  end;
end;

procedure Register;
begin
  RegisterComponents('QABook2', [TLine]);
end;

end.
```

付録FD　CHAP6¥USELINE.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。

以下のコンポーネントは、疑似的にスケール機能を使えるようにするものです。このコンポーネントは、Canvas（TCanvas型）と領域の大きさ（TRect型）から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。

付録FD　QACOMPO¥CVSCALE.PAS

```
{
        TCanvasScale:

         Canvas プロパティで Scale を利用するためのコンポーネントです。
          フォームに TCanvasScale コンポーネントを配置し、ScaleLeft、
        ScaleTop、ScaleWidth、ScaleHeight プロパティを設定します。
          描画したいキャンバス（Canvas プロパティを）と矩形領域を、
        [ ] を使って TCanvasScale コンポーネントで囲み、描画メソッドを
        呼び出します。フォームのキャンバスであれば次のようにします。

          with CanvasScale1[Canvas, ClientRect] do
          begin
            MoveTo(0, 0);
            LineTo(100, 100);
          end;

        Pen、Brush、Font プロパティも変更できますが、ペン幅はスケールの
        設定に関係なくピクセル値になります。描画できるメソッドは、ほぼ
        TCanvas と同等ですが、Polygon, CopyRect はサポートしていません。
        また、Pixels は SetPixel と GetPixel で、CopyMode は SetCopyMode
        と GetCopyMode で代用します。また、TCanvas で TRect 引数を
        取るものは、TRectReal という型に置き換える必要があります。
        TRectReal 型オブジェクトを生成するために、RealRect や
        RealBounds などの関数が用意されています。
}

unit Cvscale;

interface
```

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TCanvasScale = class;

  TPointReal = record
    X, Y: Double;
  end;

  TRectReal = record
  case Integer of
    0: (Left, Top, Right, Bottom: Double);
    1: (TopLeft, BottomRight: TPointReal);
  end;

  TScaledCanvas = object
  private
    FCanvas: TCanvas;
    FRect: TRect;
    Scale: TCanvasScale;
    Dummy: Integer;
  protected
    function Horz(x: Double): Integer;
    function HorzRev(x: Integer): Double;
    function Vert(y: Double): Integer;
    function VertRev(y: Integer): Double;
    function ToRect(const R: TRectReal): TRect;
  public
    constructor Init(ACanvas: TCanvas; ARect: TRect;
                     ACanvasScale: TCanvasScale);
    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 Polygon(const Points: array of TPointReal);}
    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;

    procedure SetCopyMode(Value: TCopyMode);
    function GetCopyMode: TCopyMode;
    procedure SetPixel(X, Y: Double; C: TColor);
    function GetPixel(X, Y: Double): TColor;
    function GetPenPos: TPointReal;
  end;

  TCanvasScale = class(TComponent)
  private
    FScaleLeft: Double;
    FScaleTop: Double;
    FScaleWidth: Double;
    FScaleHeight: Double;
    function GetScaledCanvas(C: TCanvas; R: TRect): TScaledCanvas;
    procedure SetScaleLeft(Value: Double);
    procedure SetScaleTop(Value: Double);
    procedure SetScaleWidth(Value: Double);
    procedure SetScaleHeight(Value: Double);
    procedure SetScaleRight(Value: Double);
    function GetScaleRight: Double;
    procedure SetScaleBottom(Value: Double);
    function GetScaleBottom: Double;
  public
    constructor Create(AOwner: TComponent); override;
    property Def[C: TCanvas; R: TRect]: TScaledCanvas
             read GetScaledCanvas; default;
    procedure SetScaleRect(ALeft, ATop, ARight, ABottom: Double);
    procedure SetScaleBounds(ALeft, ATop, AWidth, AHeight: Double);
  published
    property ScaleLeft: Double read FScaleLeft write SetScaleLeft;
    property ScaleTop: Double read FScaleTop write SetScaleTop;
    property ScaleWidth: Double read FScaleWidth write SetScaleWidth;
    property ScaleHeight: Double read FScaleHeight write SetScaleHeight;
    property ScaleRight: Double read GetScaleRight write SetScaleRight;
    property ScaleBottom: Double read GetScaleBottom
             write SetScaleBottom;
  end;

  function PointReal(AX, AY: Double): TPointReal;
  function RectReal(ALeft, ATop, ARight, ABottom: Double): TRectReal;
  function BoundsReal(ALeft, ATop, AWidth, AHeight: Double): TRectReal;
```

```
procedure Register;

implementation

function PointReal(AX, AY: Double): TPointReal;
begin
  with Result do
  begin
    X := AX;
    Y := AY;
  end;
end;

function RectReal(ALeft, ATop, ARight, ABottom: Double): TRectReal;
begin
  with Result do
  begin
    Left := ALeft;
    Top := ATop;
    Right := ARight;
    Bottom := ABottom;
  end;
end;

function BoundsReal(ALeft, ATop, AWidth, AHeight: Double): TRectReal;
begin
  with Result do
  begin
    Left := ALeft;
    Top := ATop;
    Right := ALeft + AWidth;
    Bottom := ATop + AHeight;
  end;
end;

function TScaledCanvas.Horz(x: Double): Integer;
begin
  with FRect, Scale do
    Result := Round((Right - Left)
              / ScaleWidth * (x - ScaleLeft)) + Left;
end;

function TScaledCanvas.HorzRev(x: Integer): Double;
begin
  with FRect, Scale do
    Result := (x - Left) * ScaleWidth / (Right - Left) + ScaleLeft;
end;

function TScaledCanvas.Vert(y: Double): Integer;
begin
```

```
  with FRect, Scale do
    Result := Round((Bottom - Top) / ScaleHeight * (y - ScaleTop)) + Top;
end;

function TScaledCanvas.VertRev(y: Integer): Double;
begin
  with FRect, Scale do
    Result := (y - Top) * ScaleHeight / (Bottom - Top) + ScaleTop;
end;

function TScaledCanvas.ToRect(const R: TRectReal): TRect;
begin
  with Result do
  begin
    Left := Horz(R.Left);
    Top := Vert(R.Top);
    Right := Horz(R.Right);
    Bottom := Vert(R.Bottom);
  end;
end;

constructor TScaledCanvas.Init(ACanvas: TCanvas; ARect: TRect;
            ACanvasScale: TCanvasScale);
begin
  FCanvas := ACanvas;
  FRect := ARect;
  Scale := ACanvasScale;
end;

procedure TScaledCanvas.Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Double);
begin
  FCanvas.Arc(Horz(X1), Vert(Y1), Horz(X2), Vert(Y2),
              Horz(X3), Vert(Y3), Horz(X4), Vert(Y4));
end;

procedure TScaledCanvas.BrushCopy(const Dest: TRectReal; Bitmap: TBitmap;
          const Source: TRectReal; Color: TColor);
begin
  FCanvas.BrushCopy(ToRect(Dest), Bitmap, ToRect(Source), Color);
end;

procedure TScaledCanvas.Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Double);
begin
  FCanvas.Chord(Horz(X1), Vert(Y1), Horz(X2), Vert(Y2),
                Horz(X3), Vert(Y3), Horz(X4), Vert(Y4));
end;

procedure TScaledCanvas.Draw(X, Y: Double; Graphic: TGraphic);
begin
  FCanvas.Draw(Horz(X), Vert(Y), Graphic);
```

```
end;

procedure TScaledCanvas.DrawFocusRect(const Rect: TRectReal);
begin
  FCanvas.DrawFocusRect(ToRect(Rect));
end;

procedure TScaledCanvas.Ellipse(X1, Y1, X2, Y2: Double);
begin
  FCanvas.Ellipse(Horz(X1), Vert(Y1), Horz(X2), Vert(Y2));
end;

procedure TScaledCanvas.FillRect(const Rect: TRectReal);
begin
  FCanvas.FillRect(ToRect(Rect));
end;

procedure TScaledCanvas.FloodFill(X, Y: Double; Color: TColor;
          FillStyle: TFillStyle);
begin
  FCanvas.FloodFill(Horz(X), Vert(Y), Color, FillStyle);
end;

procedure TScaledCanvas.FrameRect(const Rect: TRectReal);
begin
  FCanvas.FrameRect(ToRect(Rect));
end;

procedure TScaledCanvas.MoveTo(x, y: Double);
begin
  FCanvas.MoveTo(Horz(x), Vert(y));
end;

procedure TScaledCanvas.LineTo(x, y: Double);
begin
  FCanvas.LineTo(Horz(x), Vert(y));
end;

procedure TScaledCanvas.Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Double);
begin
  FCanvas.Pie(Horz(X1), Vert(Y1), Horz(X2), Vert(Y2),
              Horz(X3), Vert(Y3), Horz(X4), Vert(Y4));
end;

procedure TScaledCanvas.Rectangle(X1, Y1, X2, Y2: Double);
begin
  FCanvas.Rectangle(Horz(X1), Vert(Y1), Horz(X2), Vert(Y2));
end;

procedure TScaledCanvas.Refresh;
```

```
begin
  FCanvas.Refresh;
end;

procedure TScaledCanvas.RoundRect(X1, Y1, X2, Y2, X3, Y3: Double);
begin
  FCanvas.RoundRect(Horz(X1), Vert(Y1), Horz(X2), Vert(Y2),
                    Horz(X3), Vert(Y3));
end;

procedure TScaledCanvas.StretchDraw(const Rect: TRectReal;
          Graphic: TGraphic);
begin
  FCanvas.StretchDraw(ToRect(Rect), Graphic);
end;

function TScaledCanvas.TextHeight(const Text: string): Double;
begin
  with FRect, Scale do
    Result := FCanvas.TextHeight(Text) * ScaleHeight / (Bottom - Top);
end;

procedure TScaledCanvas.TextOut(X, Y: Double; const Text: string);
begin
  FCanvas.TextOut(Horz(X), Vert(Y), Text);
end;

procedure TScaledCanvas.TextRect(Rect: TRectReal; X, Y: Integer;
          const Text: string);
begin
  FCanvas.TextRect(ToRect(Rect), Horz(X), Vert(Y), Text);
end;

function TScaledCanvas.TextWidth(const Text: string): Double;
begin
  with FRect, Scale do
    Result := FCanvas.TextWidth(Text) * ScaleWidth / (Right - Left);
end;

function TScaledCanvas.Brush: TBrush;
begin
  Result := FCanvas.Brush;
end;

function TScaledCanvas.Font: TFont;
begin
  Result := FCanvas.Font;
end;

function TScaledCanvas.Pen: TPen;
```

```
begin
  Result := FCanvas.Pen;
end;

procedure TScaledCanvas.SetCopyMode(Value: TCopyMode);
begin
  FCanvas.CopyMode := Value;
end;

function TScaledCanvas.GetCopyMode: TCopyMode;
begin
  Result := FCanvas.CopyMode;
end;

procedure TScaledCanvas.SetPixel(X, Y: Double; C: TColor);
begin
  FCanvas.Pixels[Horz(X), Vert(Y)] := C;
end;

function TScaledCanvas.GetPixel(X, Y: Double): TColor;
begin
  Result := FCanvas.Pixels[Horz(X), Vert(Y)];
end;

function TScaledCanvas.GetPenPos: TPointReal;
var
  P: TPoint;
begin
  P := FCanvas.PenPos;
  with Result do
  begin
    X := HorzRev(P.X);
    Y := VertRev(P.Y);
  end;
end;

constructor TCanvasScale.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FScaleLeft := 0;
  FScaleTop := 0;
  if AOwner is TForm then
    with (AOwner as TForm) do
    begin
      FScaleWidth := ClientWidth;
      FScaleHeight := ClientHeight;
    end
    else
    begin
      FScaleWidth := 100;
```

```
      FScaleHeight := 100;
    end;
end;

procedure TCanvasScale.SetScaleLeft(Value: Double);
begin
  FScaleLeft := Value;
end;

procedure TCanvasScale.SetScaleTop(Value: Double);
begin
  FScaleTop := Value;
end;

procedure TCanvasScale.SetScaleWidth(Value: Double);
begin
  if Value <> 0.0 then
    FScaleWidth := Value;
end;

procedure TCanvasScale.SetScaleHeight(Value: Double);
begin
  if Value <> 0.0 then
    FScaleHeight := Value;
end;

procedure TCanvasScale.SetScaleRight(Value: Double);
begin
  if Value <> FScaleLeft then
    FScaleWidth := Value - FScaleLeft;
end;

function TCanvasScale.GetScaleRight: Double;
begin
  Result := FScaleLeft + FScaleWidth;
end;

procedure TCanvasScale.SetScaleBottom(Value: Double);
begin
  if Value <> FScaleTop then
    FScaleHeight := Value - FScaleTop;
end;

function TCanvasScale.GetScaleBottom: Double;
begin
  Result := FScaleTop + FScaleHeight;
end;

function TCanvasScale.GetScaledCanvas(C: TCanvas; R: TRect)
        : TScaledCanvas;
```

```
begin
  Result.Init(C, R, Self);
end;

procedure TCanvasScale.SetScaleRect(ALeft, ATop, ARight, ABottom:
                                    Double);
begin
  if (ALeft <> ARight) and (ATop <> ABottom) then
  begin
    FScaleLeft := ALeft;
    FScaleTop := ATop;
    FScaleWidth := ARight - ALeft;
    FScaleHeight := ABottom - ATop;
  end;
end;

procedure TCanvasScale.SetScaleBounds(ALeft, ATop, AWidth, AHeight:
                                      Double);
begin
  if (AWidth <> 0) and (AHeight <> 0) then
  begin
    FScaleLeft := ALeft;
    FScaleTop := ATop;
    FScaleWidth := AWidth;
    FScaleHeight := AHeight;
  end;
end;

procedure Register;
begin
  RegisterComponents('QABook2', [TCanvasScale]);
end;

end.
```

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, ClientRect] 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;
```

付録FD　CHAP6¥USESCALE.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。

付録FD　CHAP6¥GENERAL.PAS

```
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.
```

付録FD　CHAP6¥USEBEEP.DPR

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;
```

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 では制御変数を変更することは好ましくありません。Object Pascal では、繰返しの終了は終了値と一致するかどうかで判断されるため、終了値よりも大きい場合（downto の場合は小さい場合）は繰返し文の動作は不定となります。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。

付録FD　CHAP6¥FORSTEP.DPR

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('')はひと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C/C++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。

ヘルプ 演算子の優先順位は、オンラインヘルプの「演算子の優先順位」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 4.0でファイルに保存したデータを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 4.0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 2.0はDelphi 1.0に比べて型が追加されているため、多くの型がそのまま対応して使えます。しかし、Visual Basic 4.0のIntegerは16ビットであるのに対し、Delphi 2.0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 4.0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 4.0にはDelphi 2.0のBoolean、Extender、Compなどに対応する型はありません。

ヘルプ 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 2.0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。

重要 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 2.0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 2.0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);
  { ひと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;
```

付録FD　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 と export 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; export;
begin
  DisplayValue('Smallint', Value);
end;

procedure DisplayLongint(Value: Integer); stdcall; export;
begin
  DisplayValue('Longint', Value);
end;

procedure DisplaySingle(Value: Single); stdcall; export;
begin
  DisplayValue('Single', Value);
end;

procedure DisplayDouble(Value: Double); stdcall; export;
begin
  DisplayValue('Double', Value);
end;

procedure DisplayCurrency(Value: Currency); stdcall; export;
begin
  DisplayValue('Currency', Value);
end;

procedure DisplayString(Value: PChar); stdcall; export;
begin
  DisplayValue('String', StrPas(Value));
end;

procedure DisplayVariant(var Value: Variant); stdcall; export;
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
```

付録FD　CHAP6¥VBVALUE.DPR
CHAP6¥PASSVAR.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＋＋のプログラムと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 2.0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
VER90	コンパイラのバージョン(9.0)
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。

付録FD　CHAP7¥FACTOR.DPR

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)	

重要　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++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、矩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(:＝)は文となります。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。

ヘルプ 演算子の優先順位は、オンラインヘルプの「演算子の優先順位」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	.*、->*	(※次項目参照)
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++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。

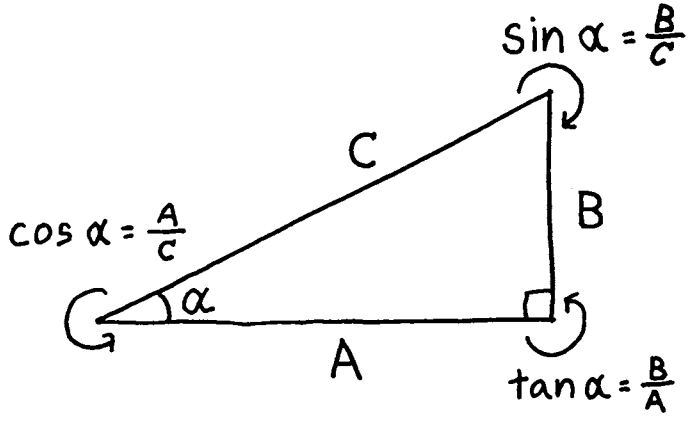
```
procedure Another;
var
  A, B: TSimple;
  fptr: s_fptr;
begin
  A := TSimple.Create;          { オブジェクトの作成 }
  B := TSimple.Create;          { オブジェクトの作成 }

  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;
```

付録FD 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 Developer 2.0やDelphi Client/Server Suite 2.0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 Desktop 2.0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レス表示（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、ゼロサプレスは、' %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。

ヘルプ　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。

付録FD　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']);
  ...
```

ヘルプ　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;
  s: string;
begin
  Result := '';
  for I := 0 to High(Params) do
  begin
    if I > 0 then
      Result := Result + ', ';
    Result := Result + VarToStr(Params[I]);
  end;
end;
```

付録FD　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、配列のひと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 2.0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;
```

ヘルプ　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「バリアントサポートルーチン」を参照してください。

付録FD　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ひと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もっとも汎用的な方法はOLEオートメーションを使うことです。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。

Delphi 2.0ではVariantによるオートメーションの制御、TAutoObjectクラスによるオートメーションサーバーの作成など、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。また、最近のBorland C++やVisual C++でもExpertやWizardなどのツールで、オートメーションに対応するソースコードを自動生成してくれるようになっています。これらを使うことで、C/C++で開発した資産をDelphiで利用することができるでしょう。

しかし、オートメーションを使う場合、お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ためには、DLLを使うことが望ましく、.OBJファイルのスタティックリンクでは対応が難しくなります。

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 2.0とBorland C++ 5.0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 2.0とBorland C++ 5.0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 2.0	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を使うべきでしょう。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。

付録FD　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、[ﾌﾟﾛｼﾞｪｸﾄ(P)]-[ｵﾌﾟｼｮﾝ(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; export;
procedure FreePerson(APerson: TPerson); stdcall; export;
```

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);
};
```

付録FD　CHAP7¥RICHED.DPR
CHAP7¥MAKEEXE.BAT

付　録

付録ディスクについて
用語解説

付録ディスクについて

付録ディスクに含まれるプログラムは、「Delphi 2.0 Q&A 120選」の理解を助けるためのプログラムです。ファイルは、QABOOK2ディレクトリ(フォルダ)の中にあります。

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、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。したがって、すべて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一度に登録するには、Delphiの［ｺﾝﾎﾟｰﾈﾝﾄ(C)|ｲﾝｽﾄｰﾙ(I)］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、¥QABOOK2¥QACOMPOディレクトリのQACOMPO.PASを追加します。個々のコンポーネントを必要に応じて登録したい場合は、QACOMPO.PAS以外のコンポーネントソース(.PAS)を登録します。QACOMPO.PASと他のコンポーネントは同時には登録できません。

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、コンポーネントユニットの内容を次ページ以降に示します(ディレクトリQABOOK2の記述は省略してあります)。

オンラインヘルプ(QABOOK2.HLP)の使い方

¥QABOOK2¥HELPディレクトリにあるQABOOK2.HLPは、「Delphi 2.0 Q&A 120選」に含まれているQ&Aの内容を、そのままオンラインヘルプで参照できるようにしたものです。このディレクトリのINSTHELP.EXEを使うと簡単にヘルプファイルのコピーと設定ができます。手作業でファイルを組み込む場合は、QABOOK2.HLPおよびQABOOK2.CNTをDelphi 2.0をインストールしたHELPサブディレクトリ(デフォルトでは¥Program Files¥Borland¥Delphi 2.0¥Help)にコピーし、このディレクトリのDELPHLCNTの末尾に「:Include qabook2.cnt」という記述を追加してください。

ファイル名	ページ	内容
(第1章 統合開発環境)		
CHAP1¥SMALLAPR.DPR	24	小さいアプリケーション
CHAP1¥CPUVIEW.DPR	28	CPUウィンドウの設定
(第2章 プログラミング)		
CHAP2¥ANGLESTR.DPR	51	文字列を斜めに描画
CHAP2¥BLUEFRM.DPR	34	ビットマップの代入
CHAP2¥CHGPROP.DPR	31	プロパティの変更
CHAP2¥CHKCHAR.DPR	47	文字判定ルーチン
CHAP2¥CONAPP.DPR	44	コンソールアプリケーション
CHAP2¥CVMSBIN.DPR	57	MSBIN形式の変換
CHAP2¥DESKBMP.DPR	33	スクリーン全体のビットマップ
CHAP2¥ENUMWIN.DPR	53	コールバック関数
CHAP2¥EXECCMD.DPR	37	実行ファイルの呼び出し
CHAP2¥EXITWIN.DPR	40	Windowsの再起動
CHAP2¥IDENTFIL.DPR	54	ファイルアクセス
CHAP2¥IFELSE.DPR	30	セミコロンの付け方
CHAP2¥IOPORT.DPR	45	I/Oポートへの入出力
CHAP2¥MKMETAS.DPR	36	メタファイルの作成
CHAP2¥PRTEST.DPR	41	印字フォントの指定
	42	イメージの印刷
CHAP2¥PRTEXT.DPR	41	テキストの印刷
CHAP2¥RBUTTONS.DPR	33	複数のラジオボタンのグループ
CHAP2¥SCONST.DPR	31	文字列定数におけるシングルクォート
CHAP2¥SETFORE.DPR	32	アクティブなウィンドウの設定
CHAP2¥VIEWINLDPR	54	ファイルアクセス
CHAP2¥WFOCUS.DPR	32	重複する識別子の参照方法
CHAP2¥WINDIR.DPR	58	バッファとしての文字列型
(第3章 アプリケーション/フォーム)		
CHAP3¥APPICON.DPR	85	アプリケーションのアイコン
CHAP3¥CHGCSR.DPR	83	カーソルの変更
CHAP3¥CMDSHOWP.DPR	73	アイコン化で実行
CHAP3¥DISPARG.DPR	65	コマンドライン引数
CHAP3¥DRGFILE.DPR	93	ドラッグアンドドロップできるフォーム

ファイル名	ページ	内容
CHAP3¥DYNFORM.DPR	66	選択可能なフォーム
CHAP3¥ELLIPFRM.DPR	88	矩形でないフォーム
CHAP3¥ENDSESS.DPR	89	Windows終了の検出
CHAP3¥EXEPATH.DPR	64	起動ディレクトリパス
CHAP3¥FIXPOS.DPR	90	動かせないフォーム
CHAP3¥FMSCALE.DPR	75	フォームのスケール
CHAP3¥FRMPROJ.DPR	60	複数のフォームを使うプロジェクト
CHAP3¥KEEPMIN.DPR	94	フォームのアイコン状態を維持
CHAP3¥MDIHELP.DPR	91	MDIとヘルプの呼び出し
CHAP3¥MDIPROG.DPR	68	ShowModalとフォームメモリの解放
CHAP3¥NOTITLE.DPR	86	タイトルバーのないフォーム
	87	フォームの移動
CHAP3¥ONLYONE.DPR	62	二重起動の禁止
CHAP3¥ORIGCSR.DPR	84	オリジナルのカーソル
CHAP3¥PROGRESS.DPR	69	オープニングダイアログ
CHAP3¥SCRLPOS.DPR	77	フォームのスクロール位置
CHAP3¥SPLASH.DPR	69	オープニングダイアログ
CHAP3¥USEWPLC.DPR	78	レジストリの使い方
	79	フォームの位置や大きさの保存
(第4章 コンポーネント)		
CHAP4¥ACTPAGE.DPR	136	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	108	大規模テキストの編集
	109	テキストの編集とカーソル位置の移動
	110	テキストの編集と上書きモード
	111	テキストの編集と文字列検索
CHAP4¥CALCDEMO.DPR	111	Editコンポーネントと右寄せ表示
CHAP4¥DRAWIMG.DPR	123	Imageコンポーネントへの描画
CHAP4¥DYNEDIT.DPR	116	コンポーネントの動的作成
CHAP4¥EDITMOV.DPR	103	矢印キーでコンポーネントを移動する
CHAP4¥HINTWIN.DPR	119	ヒント表示を変更する
CHAP4¥ODLBOX.DPR	106	リストボックスの選択項目の色指定
CHAP4¥RESIZE.DPR	102	実行時のパネルの大きさの変更
CHAP4¥SGRIDML.DPR	100	文字列グリッドと複数行の表示

ファイル名	ページ	内容
CHAP4 ¥ STRGRID.DPR	97	文字列グリッドの選択セルの色指定
	100	文字列グリッドのセル単位での色指定
	101	文字列グリッドと固定セルのクリック
CHAP4 ¥ USECOMPO.DPR	132	ヨミガナの取り出し
CHAP4 ¥ USEDRGM.DPR	129	Memo コンポーネントとドラッグアンドドロップ
CHAP4 ¥ USEPBLST.DPR	126	コンポーネントのプロパティと TStrings 型
CHAP4 ¥ USETEXT.DPR	112	ラベルのZオーダー
CHAP4 ¥ ZOOMPNL.DPR	96	コンポーネントの描画
CHAP4 ¥ ZORDER.DPR	115	実行時のZオーダーの変更、確認
(第5章 データベース)		
CHAP5 ¥ CONVTXT.DPR	155	テーブル形式の変換
CHAP5 ¥ CSVTODB.DPR	156	テキスト形式のデータを変換
CHAP5 ¥ DBALIAS.DPR	161	ユーザー定義のデータベースエリアス
CHAP5 ¥ DBGMULT.DPR	141	DBGrid と複数のテーブル表示
CHAP5 ¥ DBGMREC.DPR	143	DBGrid と複数レコードの選択
CHAP5 ¥ DRAWCELL.DPR	139	DBGrid と選択中のセル描画
CHAP5 ¥ DYNTABLE.DPR	145	テーブルコンポーネントを実行時に作成する
CHAP5 ¥ FINDREC.DPR	162	検索の高速化
CHAP5 ¥ MKTABLE.DPR	149	新しいテーブルの作成
	152	テーブルにインデックスを付ける
CHAP5 ¥ NAVDSET.DPR	163	DataSource を使ったレコード操作
CHAP5 ¥ TBLRANGE.DPR	158	テーブルの範囲指定
	160	テーブルの範囲指定
CHAP5 ¥ USEDBEX.DPR	147	DBCtrlGrid と DBImage/DBMemo の表示
CHAP5 ¥ USENAVB.DPR	164	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	177	Visual Basic の AutoRedraw プロパティの代用
CHAP6 ¥ CTLARRY.DPR	172	Visual Basic のコントロール配列の代用
CHAP6 ¥ FORSTEP.DPR	198	Visual Basic の For~Step の代用
CHAP6 ¥ MKVBDAT.VBP	201	Visual Basic のデータの利用
CHAP6 ¥ PASSVAR.VBP	213	Visual Basic プログラムとの連携
CHAP6 ¥ PROCMSG.DPR	170	Visual Basic の DoEvents の代用
CHAP6 ¥ RDVBDAT.DPR	201	Visual Basic のデータの利用
CHAP6 ¥ USEBEEP.DPR	196	Visual Basic のジェネラルプロシージャの代用

ファイル名	ページ	内容
CHAP6¥USELINE.DPR	179	ライン(直線)コントロール
CHAP6¥USESCALE.DPR	185	Visual Basicのスケール機能の代用
CHAP6¥VBVALUE.DPR	213	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	229	書式付き文字列処理
CHAP7¥DYNARRY.DPR	233	動的配列の確保
CHAP7¥FACTOR.DPR	219	C/C++のreturn文の代替
CHAP7¥MAKEDLL.BAT	236	C/C++資産の活用
CHAP7¥MAKEEXE.BAT	239	C/C++からDelphiフォームの呼び出し
CHAP7¥METHODP.DPR	225	メンバへのポインタ
CHAP7¥OPENARR.DPR	230	可変個引数を使った手続き
CHAP7¥RICHED.DPR	239	C/C++からDelphiフォームの呼び出し
CHAP7¥USECMOD.DPR	236	C/C++資産の活用
(オリジナルコンポーネント)		
QACOMPO¥COMPOSTR.PAS	132	ヨミガナの取得
QACOMPO¥CVSCALE.PAS	185	Canvasに対するスケール設定
QACOMPO¥DBBLOBEX.PAS	147	DBCtrlGridで使えるDBMemo/DBImage
QACOMPO¥DBNAVBTN.PAS	164	レコード操作用の個別ボタン
QACOMPO¥DRAGMEMO.PAS	129	ドラッグアンドドロップ対応のMemo
QACOMPO¥LINE.PAS	179	ライン(直線)コントロール
QACOMPO¥PBOXLST.PAS	126	コンポーネントのプロパティとTStrings型
QACOMPO¥TEXTCTRL.PAS	112	ラベルのZオーダー
QACOMPO¥QACOMPO.PAS		すべてのコンポーネントをまとめて登録する
(ヘルプファイル)		
HELP¥INSTHELP.EXE		QABOOK2.HLPインストールプログラム
HELP¥QABOOK2.HLP		すべてのQ&Aを収録したヘルプファイル
HELP¥QABOOK2.CNT		QABOOK2.HLPのためのcontentsファイル

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。exportという予約語をつけて定義する。

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。

INDEX

アルファベット

A・B・C

D・F・G

M・N・P・R

S・W・Z

五十音

あ

か

ま

や

ら

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 2.0 Q&A 120選

1996年12月23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
イラスト　lica
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　シナノ印刷株式会社

定価はカバーに表示してあります。

小社の出版情報をインターネットで紹介しています。
http://www.villagecenter.co.jp/book.html

ISBN4-89436-044-6　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
120個のQ&Aを取り上げています。

■付録ディスク

本書の全ての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、約100個の完結したサンプルプロ
グラムやコンポーネントが含まれています。

- テーブルの印刷とプレビュー機能
- メタファイルの作成
- ヒント表示の拡張
- 独自の配色を使った文字列グリッド
- テーブル、インデックスの作成
- テキストデータとテーブルの変換
- VBデータの利用
- CプログラムからDLLの作成
- ラインコンポーネント

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120個のQ&Aを取り上げています。

■付録ディスク

本書の全ての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、約100個の完結したサンプルプログラムやコンポーネントが含まれています。

- テーブルの印刷とプレビュー機能
- メタファイルの作成
- ヒント表示の拡張
- 独自の配色を使った文字列グリッド
- テーブル、インデックスの作成
- テキストデータとテーブルの変換
- VBデータの利用
- Cプログラムから利用できるDLLの作成
- ラインコンポーネント

ISBN4-89436-044-6 C3055 P1800E 定価1,800円(本体1,748円)