Delphi Language Guide

Delphi for Microsoft Win32
Delphi for the Microsoft .NET Framework

Borland®
Excellence Endures

Borland Software Corporation
100 Enterprise Way
Scotts Valley, California 95066-3249
www.borland.com

Refer to the file deploy.html for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

October 2004

PDF

Reference

Delphi Language Guide

Data Types, Variables, and Constants

Procedures and Functions

Classes and Objects

Standard Routines and I/O

Libraries and Packages

Object Interfaces

Memory Management

Program Control

Inline Assembly Code (Win32 Only)

.NET Topics

Language Guide

Delphi

Delphi Language Guide

The Delphi Language guide describes the Delphi language as it is used in Borland development tools. This book describes the Delphi language on both the Win32, and .NET development platforms. Specific differences in the language between the two platforms are marked as appropriate.

Language Overview

Delphi is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Based on Object Pascal, its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming. Delphi has special features that support Borland's component framework and RAD environment. For the most part, descriptions and examples in this language guide assume that you are using Borland development tools.

Most developers using Borland software development tools write and compile their code in the integrated development environment (IDE). Borland development tools handle many details of setting up projects and source files, such as maintenance of dependency information among units. The product also places constraints on program organization that are not, strictly speaking, part of the Object Pascal language specification. For example, Borland development tools enforce certain file- and program-naming conventions that you can avoid if you write your programs outside of the IDE and compile them from the command prompt.

This language guide generally assumes that you are working in the IDE and that you are building applications that use the Borland Visual Component Library (VCL). Occasionally, however, Delphi-specific rules are distinguished from rules that apply to all Object Pascal programming. This text covers both the Win32 Delphi language compiler, and the Delphi for .NET language compiler. Platform-specific language differences and features are noted where necessary.

This section covers the following topics:

- Program Organization. Covers the basic language features that allow you to partition your application into units and namespaces.
- Example Programs. Small examples of both console and GUI applications are shown, with basic instructions on running the compiler from the command-line.

Program Organization

Delphi programs are usually divided into source-code modules called units. Most programs begin with a program heading, which specifies a name for the program. The program heading is followed by an optional uses clause, then a block of declarations and statements. The uses clause lists units that are linked into the program; these units, which can be shared by different programs, often have uses clauses of their own.

The uses clause provides the compiler with information about dependencies among modules. Because this information is stored in the modules themselves, most Delphi language programs do not require makefiles, header files, or preprocessor "include" directives.

Delphi Source Files

The compiler expects to find Delphi source code in files of three kinds:

- Unit source files (which end with the .pas extension)
- Project files (which end with the .dpr extension)
- Package source files (which end with the .dpk extension)

Unit source files typically contain most of the code in an application. Each application has a single project file and several unit files; the project file, which corresponds to the program file in traditional Pascal, organizes the unit files into an application. Borland development tools automatically maintain a project file for each application.

If you are compiling a program from the command line, you can put all your source code into unit (.pas) files. If you use the IDE to build your application, it will produce a project (.dpr) file.

Package source files are similar to project files, but they are used to construct special dynamically linkable libraries called packages.

Other Files Used to Build Applications

In addition to source-code modules, Borland products use several non-Pascal files to build applications. These files are maintained automatically by the IDE, and include

- VCL form files (which have a .dfm extension on Win32, and .nfm on .NET)
- Resource files (which end with .res)
- Project options files (which end with .dof)

A VCL form file contains the description of the properties of the form and the components it owns. Each form file represents a single form, which usually corresponds to a window or dialog box in an application. The IDE allows you to view and edit form files as text, and to save form files as either text (a format very suitable for version control) or binary. Although the default behavior is to save form files as text, they are usually not edited manually; it is more common to use Borland's visual design tools for this purpose. Each project has at least one form, and each form has an associated unit (.pas) file that, by default, has the same name as the form file.

In addition to VCL form files, each project uses a resource (.res) file to hold the application's icon and other resources such as strings. By default, this file has the same name as the project (.dpr) file.

A project options (.dof) file contains compiler and linker settings, search path information, version information, and so forth. Each project has an associated project options file with the same name as the project (.dpr) file. Usually, the options in this file are set from Project Options dialog.

Various tools in the IDE store data in files of other types. Desktop settings (.dsk) files contain information about the arrangement of windows and other configuration options; desktop settings can be project-specific or environment-wide. These files have no direct effect on compilation.

Compiler-Generated Files

The first time you build an application or a package, the compiler produces a compiled unit file (.dcu on Win32, .dcuil on .NET) for each new unit used in your project; all the .dcu/.dcuil files in your project are then linked to create a single executable or shared package. The first time you build a package, the compiler produces a file for each new unit contained in the package, and then creates both a .dcp and a package file.If you use the **GD** switch, the linker generates a map file and a .drc file; the .drc file, which contains string resources, can be compiled into a resource file.

When you build a project, individual units are not recompiled unless their source (.pas) files have changed since the last compilation, their .dcu/.dpu files cannot be found, you explicitly tell the compiler to reprocess them, or the interface of the unit depends on another unit which has been changed. In fact, it is not necessary for a unit's source file to be present at all, as long as the compiler can find the compiled unit file and that unit has no dependencies on other units that have changed.

Example Programs

The examples that follow illustrate basic features of Delphi programming. The examples show simple applications that would not normally be compiled from the IDE; you can compile them from the command line.

A Simple Console Application

The program below is a simple console application that you can compile and run from the command prompt.

```
program greeting;

  {$APPTYPE CONSOLE}

  var MyMessage string;
```

```
    begin
         MyMessage := 'Hello world!';
   Writeln(MyMessage);
 end.
```

The first line declares a program called Greeting. The `{$APPTYPE CONSOLE}` directive tells the compiler that this is a console application, to be run from the command line. The next line declares a variable called MyMessage, which holds a string. (Delphi has genuine string data types.) The program then assigns the string "Hello world!" to the variable MyMessage, and sends the contents of MyMessage to the standard output using the `Writeln` procedure. (`Writeln` is defined implicitly in the `System` unit, which the compiler automatically includes in every application.)

You can type this program into a file called greeting.pas or greeting.dpr and compile it by entering

dcc32 greeting

to produce a Win32 executable, or

dccil greeting

to produce a managed .NET executable. In either case, the resulting executable prints the message Hello world!

Aside from its simplicity, this example differs in several important ways from programs that you are likely to write with Borland development tools. First, it is a console application. Borland development tools are most often used to write applications with graphical interfaces; hence, you would not ordinarily call `Writeln`. Moreover, the entire example program (save for `Writeln`) is in a single file. In a typical GUI application, the program heading the first line of the example would be placed in a separate project file that would not contain any of the actual application logic, other than a few calls to routines defined in unit files.

A More Complicated Example

The next example shows a program that is divided into two files: a project file and a unit file. The project file, which you can save as greeting.dpr, looks like this:

```
program greeting;

  {$APPTYPE CONSOLE}

     uses Unit1;

        begin
          PrintMessage('Hello World!');
  end.
```

The first line declares a program called `greeting`, which, once again, is a console application. The `uses Unit1;` clause tells the compiler that the program `greeting` depends on a unit called `Unit1`. Finally, the program calls the `PrintMessage` procedure, passing to it the string Hello World! The `PrintMessage` procedure is defined in `Unit1`. Here is the source code for `Unit1`, which must be saved in a file called Unit1.pas:

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation;
```

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

end.
```

Unit1 defines a procedure called PrintMessage that takes a single string as an argument and sends the string to the standard output. (In Delphi, routines that do not return a value are called procedures. Routines that return a value are called functions.) Notice that PrintMessage is declared twice in Unit1. The first declaration, under the reserved word interface, makes PrintMessage available to other modules (such as greeting) that use Unit1. The second declaration, under the reserved word implementation, actually defines PrintMessage.

You can now compile Greeting from the command line by entering

dcc32 greeting

to produce a Win32 executable, or

dccil greeting

to produce a managed .NET executable.

There is no need to include Unit1 as a command-line argument. When the compiler processes greeting.dpr, it automatically looks for unit files that the greeting program depends on. The resulting executable does the same thing as our first example: it prints the message Hello world!

A VCL Application

Our next example is an application built using the Visual Component Library (VCL) components in the IDE. This program uses automatically generated form and resource files, so you won't be able to compile it from the source code alone. But it illustrates important features of the Delphi Language. In addition to multiple units, the program uses classes and objects

The program includes a project file and two new unit files. First, the project file:

```
program greeting;

        uses Forms, Unit1, Unit2;
        {$R *.res} // This directive links the project's resource file.

begin
   // Calls to global Application instance
   Application.Initialize;
   Application.CreateForm(TForm1, Form1);
   Application.CreateForm(TForm2, Form2);
   Application.Run;
end.
```

Once again, our program is called greeting. It uses three units: Forms, which is part of VCL; Unit1, which is associated with the application's main form (Form1); and Unit2, which is associated with another form (Form2).

The program makes a series of calls to an object named Application, which is an instance of the TApplication class defined in the Forms unit. (Every project has an automatically generated Application object.) Two of these calls invoke a TApplication method named CreateForm. The first call to CreateForm creates Form1, an instance of the TForm1 class defined in Unit1. The second call to CreateForm creates Form2, an instance of the TForm2 class defined in Unit2.

`Unit1` looks like this:

```
unit Unit1;

interface

uses SysUtils, Types, Classes, Graphics, Controls, Forms, Dialogs;

type

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

var
  Form1: TForm1;

implementation

uses Unit2;

{$R *.dfm}

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

end.
```

`Unit1` creates a class named `TForm1` (derived from TForm) and an instance of this class, `Form1`. `TForm1` includes a `buttonButton1`, an instance of TButton and a procedure named `Button1Click` that is called when the user presses `Button1`. `Button1Click` hides `Form1` and it displays `Form2` (the call to `Form2.ShowModal`).

Note: In the previous example, `Form2.ShowModal` relies on the use of auto-created forms. While this is fine for example code, using auto-created forms is actively discouraged.

`Form2` is defined in `Unit2`:

```
unit Unit2;

interface

uses SysUtils, Types, Classes, Graphics, Controls, Forms, Dialogs;

type
TForm2 = class(TForm)
  Label1: TLabel;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
end;

var
  Form2: TForm2;

implementation
```

```
uses Unit1;

{$R *.dfm}

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

end.
```

Unit2 creates a class named TForm2 and an instance of this class, Form2. TForm2 includes a button (CancelButton, an instance of TButton) and a label (Label1, an instance of TLabel). You can't see this from the source code, but Label1 displays a caption that reads Hello world! The caption is defined in Form2's form file, Unit2.dfm.

TForm2 declares and defines a method CancelButtonClick which will be invoked at runtime whenever the user presses CancelButton. This procedure (along with Unit1's TForm1.Button1Click) is called an *event handler* because it responds to events that occur while the program is running. Event handlers are assigned to specific events by the form files for Form1 and Form2.

When the greeting program starts, Form1 is displayed and Form2 is invisible. (By default, only the first form created in the project file is visible at runtime. This is called the project's main form.) When the user presses the button on Form1, Form2, displays the Hello world! greeting. When the user presses the CancelButton or the Close button on the title bar, Form2 closes.

Programs and Units

A Delphi program is constructed from source code modules called units. The units are tied together by a special source code module that contains either the program, library, or package header. Each unit is stored in its own file and compiled separately; compiled units are linked to create an application. Delphi 2005 introduces hierarchical namespaces, giving you even more flexibility in organizing your units. Namespaces and units allow you to

- Divide large programs into modules that can be edited separately.
- Create libraries that you can share among programs.
- Distribute libraries to other developers without making the source code available.

This topic covers the overall structure of a Delphi application: the program header, unit declaration syntax, and the uses clause. Specific differences between the Win32 and .NET platforms are noted in the text. The Delphi compiler does not support .NET namespaces on the Win32 platform. The Delphi 2005 compiler does support hierarchical .NET namespaces; this topic is covered in the following section, Using Namespaces with Delphi.

Program Structure and Syntax

A complete, executable Delphi application consists of multiple unit modules, all tied together by a single source code module called a project file. In traditional Pascal programming, all source code, including the main program, is stored in .pas files. Borland tools use the file extension .dpr to designate the main program source module, while most other source code resides in unit files having the traditional .pas extension. To build a project, the compiler needs the project source file, and either a source file or a compiled unit file for each unit.

Note: Strictly speaking, you need not explicitly use any units in a project, but all programs automatically use the `System` unit and the `SysInit` unit.

The source code file for an executable Delphi application contains

- a program heading,
- a uses clause (optional), and
- a block of declarations and executable statements.

Additionally, a Delphi 2005 program may contain a namespaces clause, to specify additional namespaces in which to search for generic units. This topic is covered in more detail in the section Using .NET Namespaces with Delphi.

The compiler, and hence the IDE, expect to find these three elements in a single project (.dpr) file.

The Program Heading

The program heading specifies a name for the executable program. It consists of the reserved word program, followed by a valid identifier, followed by a semicolon. The identifier must match the project source file name.

The following example shows the project source file for a program called Editor. Since the program is called Editor, this project file is called Editor.dpr.

```
program Editor;

  uses Forms, REAbout, // An "About" box
       REMain;         // Main form

  {$R *.res}

  begin
   Application.Title := 'Text Editor';
```

```
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
 end.
```

The first line contains the program heading. The uses clause in this example specifies a dependency on three additional units: `Forms`, `REAbout`, and `REMain`. The `$R` compiler directive links the project's resource file into the program. Finally, the block of statements between the begin and end keywords are executed when the program runs. The project file, like all Delphi source files, ends with a period (not a semicolon).

Delphi project files are usually short, since most of a program's logic resides in its unit files. A Delphi project file typically contains only enough code to launch the application's main window, and start the event processing loop. Project files are generated and maintained automatically by the IDE, and it is seldom necessary to edit them manually.

In standard Pascal, a program heading can include parameters after the program name:

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

Borland's Delphi ignores these parameters.

In Delphi 2005, a the program heading introduces its own namespace, which is called the project default namespace. This is also true for the library and package headers, when these types of projects are compiled for the .NET platform.

The Program Uses Clause

The uses clause lists those units that are incorporated into the program. These units may in turn have uses clauses of their own. For more information on the uses clause within a unit source file, see *Unit References and the Uses Clause*, below.

The uses clause consists of the keyword uses, followed by a comma delimited list of units the project file directly depends on.

The Block

The block contains a simple or structured statement that is executed when the program runs. In most program files, the block consists of a compound statement bracketed between the reserved words begin and end, whose component statements are simply method calls to the project's `Application` object. Most projects have a global `Application` variable that holds an instance of TApplication, TWebApplication, or TServiceApplication. The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block.

Unit Structure and Syntax

A unit consists of types (including classes), constants, variables, and routines (functions and procedures). Each unit is defined in its own source (.pas) file.

A unit file begins with a unit heading, which is followed by the interface keyword. Following the interface keyword, the uses clause specifies a list of unit dependencies. Next comes the implementation section, followed by the optional initialization, and finalization sections. A skeleton unit source file looks like this:

```
unit Unit1;

interface

uses // List of unit dependencies goes here...
```

```
implementation

uses // List of unit dependencies goes here...

// Implementation of class methods, procedures, and functions goes here...

initialization

// Unit initialization code goes here...

finalization

// Unit finalization code goes here...

end.
```

The unit must conclude with the reserved word end followed by a period.

The Unit Heading

The unit heading specifies the unit's name. It consists of the reserved word unit, followed by a valid identifier, followed by a semicolon. For applications developed using Borland tools, the identifier must match the unit file name. Thus, the unit heading

```
unit MainForm;
```

would occur in a source file called MainForm.pas, and the file containing the compiled unit would be MainForm.dcu or MainForm.dpu.

Unit names must be unique within a project. Even if their unit files are in different directories, two units with the same name cannot be used in a single program.

The Interface Section

The interface section of a unit begins with the reserved word interface and continues until the beginning of the implementation section. The interface section declares constants, types, variables, procedures, and functions that are available to clients. That is, to other units or programs that wish to use elements from this unit. These entities are called *public* because code in other units can access them as if they were declared in the unit itself.

The interface declaration of a procedure or function includes only the routine's signature. That is, the routine's name, parameters, and return type (for functions). The block containing executable code for the procedure or function follows in the implementation section. Thus procedure and function declarations in the interface section work like forward declarations.

The interface declaration for a class must include declarations for all class members: fields, properties, procedures, and functions.

The interface section can include its own uses clause, which must appear immediately after the keyword interface.

The Implementation Section

The implementation section of a unit begins with the reserved word implementation and continues until the beginning of the initialization section or, if there is no initialization section, until the end of the unit. The implementation section defines procedures and functions that are declared in the interface section. Within the implementation section, these procedures and functions may be defined and called in any order. You can omit parameter lists from public procedure and function headings when you define them in the implementation section; but if you include a parameter list, it must match the declaration in the interface section exactly.

In addition to definitions of public procedures and functions, the implementation section can declare constants, types (including classes), variables, procedures, and functions that are *private* to the unit. That is, unlike the interface section, entities declared in the implementation section are inaccessible to other units.

The implementation section can include its own uses clause, which must appear immediately after the keyword implementation. The identifiers declared within units specified in the implementation section are only available for use within the implementation section itself. You cannot refer to such identifiers in the interface section.

The Initialization Section

The initialization section is optional. It begins with the reserved word initialization and continues until the beginning of the finalization section or, if there is no finalization section, until the end of the unit. The initialization section contains statements that are executed, in the order in which they appear, on program start-up. So, for example, if you have defined data structures that need to be initialized, you can do this in the initialization section.

For units in the interfaceuses list, the initialization sections of the units used by a client are executed in the order in which the units appear in the client's uses clause.

The Finalization Section

The finalization section is optional and can appear only in units that have an initialization section. The finalization section begins with the reserved word finalization and continues until the end of the unit. It contains statements that are executed when the main program terminates (unless the *Halt* procedure is used to terminate the program). Use the finalization section to free resources that are allocated in the initialization section.

Finalization sections are executed in the opposite order from initialization sections. For example, if your application initializes units A, B, and C, in that order, it will finalize them in the order C, B, and A.

Once a unit's initialization code starts to execute, the corresponding finalization section is guaranteed to execute when the application shuts down. The finalization section must therefore be able to handle incompletely initialized data, since, if a runtime error occurs, the initialization code might not execute completely.

Note: The initialization and finalization sections behave differently when code is compiled for the managed .NET environment. See the chapter on Memory Management for more information.

Unit References and the Uses Clause

A uses clause lists units used by the program, library, or unit in which the clause appears. A uses clause can occur in

- the project file for a program, or library
- the interface section of a unit
- the implementation section of a unit

Most project files contain a uses clause, as do the interface sections of most units. The implementation section of a unit can contain its own uses clause as well.

The `System` unit and the `SysInit` unit are used automatically by every application and cannot be listed explicitly in the uses clause. (`System` implements routines for file I/O, string handling, floating point operations, dynamic memory allocation, and so forth.) Other standard library units, such as `SysUtils`, must be explicitly included in the uses clause. In most cases, all necessary units are placed in the uses clause by the IDE, as you add and remove units from your project.

In unit declarations and uses clauses, unit names must match the file names in case. In other contexts (such as qualified identifiers), unit names are case insensitive. To avoid problems with unit references, refer to the unit source file explicitly:

```
uses MyUnit in "myunit.pas";
```

If such an explicit reference appears in the project file, other source files can refer to the unit with a simple uses clause that does not need to match case:

```
uses Myunit;
```

The Syntax of a Uses Clause

A uses clause consists of the reserved word uses, followed by one or more comma delimited unit names, followed by a semicolon. Examples:

```
uses Forms, Main;

uses
   Forms,
   Main;

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

In the uses clause of a program or library, any unit name may be followed by the reserved word in and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. Examples:

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

Use the keyword in after a unit name when you need to specify the unit's source file. Since the IDE expects unit names to match the names of the source files in which they reside, there is usually no reason to do this. Using in is necessary only when the location of the source file is unclear, for example when

- You have used a source file that is in a different directory from the project file, and that directory is not in the compiler's search path.
- Different directories in the compiler's search path have identically named units.
- You are compiling a console application from the command line, and you have named a unit with an identifier that doesn't match the name of its source file.

The compiler also relies on the in ... construction to determine which units are part of a project. Only units that appear in a project (.dpr) file's uses clause followed by in and a file name are considered to be part of the project; other units in the uses clause are used by the project without belonging to it. This distinction has no effect on compilation, but it affects IDE tools like the **Project Manager**.

In the uses clause of a unit, you cannot use in to tell the compiler where to find a source file. Every unit must be in the compiler's search path. Moreover, unit names must match the names of their source files.

Multiple and Indirect Unit References

The order in which units appear in the uses clause determines the order of their initialization and affects the way identifiers are located by the compiler. If two units declare a variable, constant, type, procedure, or function with the

same name, the compiler uses the one from the unit listed last in the uses clause. (To access the identifier from the other unit, you would have to add a qualifier: `UnitName.Identifier`.)

A uses clause need include only units used directly by the program or unit in which the clause appears. That is, if unit A references constants, types, variables, procedures, or functions that are declared in unit B, then A must use B explicitly. If B in turn references identifiers from unit C, then A is indirectly dependent on C; in this case, C needn't be included in a uses clause in A, but the compiler must still be able to find both B and C in order to process A.

The following example illustrates indirect dependency.

```
program Prog;
uses Unit2;
const a = b;
// ...

unit Unit2;
interface
uses Unit1;
const b = c;
// ...

unit Unit1;
interface
const c = 1;
// ...
```

In this example, `Prog` depends directly on `Unit2`, which depends directly on `Unit1`. Hence `Prog` is indirectly dependent on `Unit1`. Because `Unit1` does not appear in `Prog`'s **uses** clause, identifiers declared in `Unit1` are not available to `Prog`.

To compile a client module, the compiler needs to locate all units that the client depends on, directly or indirectly. Unless the source code for these units has changed, however, the compiler needs only their .dcu (Win32) or .dcuil (.NET) files, not their source (.pas) files.

When a change is made in the interface section of a unit, other units that depend on the change must be recompiled. But when changes are made only in the implementation or other sections of a unit, dependent units don't have to be recompiled. The compiler tracks these dependencies automatically and recompiles units only when necessary.

Circular Unit References

When units reference each other directly or indirectly, the units are said to be mutually dependent. Mutual dependencies are allowed as long as there are no circular paths connecting the uses clause of one interface section to the uses clause of another. In other words, starting from the interface section of a unit, it must never be possible to return to that unit by following references through interface sections of other units. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the uses clause of at least one implementation section.

In the simplest case of two mutually dependent units, this means that the units cannot list each other in their interface uses clauses. So the following example leads to a compilation error:

```
unit Unit1;
interface
uses Unit2;
// ...

unit Unit2;
interface
```

```
uses Unit1;
// ...
```

However, the two units can legally reference each other if one of the references is moved to the implementation section:

```
unit Unit1;
interface
uses Unit2;
// ...

unit Unit2;
interface
//...

implementation
uses Unit1;
// ...
```

To reduce the chance of circular references, it's a good idea to list units in the implementation uses clause whenever possible. Only when identifiers from another unit are used in the interface section is it necessary to list that unit in the interface uses clause.

Using Namespaces with Delphi

In Delphi, a unit is the basic container for types. Microsoft's Common Language Runtime (CLR) introduces another layer of organization called a namespace. In the .NET Framework, a namespace is a conceptual container of types. In Delphi, a namespace is a container of Delphi units. The addition of namespaces gives Delphi the ability to access and extend classes in the .NET Framework.

Unlike traditional Delphi units, namespaces can be nested to form a containment hierarchy. Nested namespaces provide a way to organize identifiers and types, and are used to disambiguate types with the same name. Since they are a container for Delphi units, namespaces may also be used to differentiate between units of the same name, that reside in different packages.

For example, the class `MyClass` in `MyNameSpace`, is different from the class `MyClass` in `YourNamespace`. At runtime, the CLR always refers to classes and types by their fully qualified names: the assembly name, followed by the namespace that contains the type. The CLR itself has no concept or implementation of the namespace hierarchy; it is purely a notational convenience of the programming language.

The following topics are covered:

- Project default namespaces, and namespace declaration.
- Namespace search scope.
- Using namespaces in Delphi units.

Declaring Namespaces

In Delphi 2005, a project file (program, library, or package) implicitly introduces its own namespace, called the *project default namespace*. A unit may be a member of the project default namespace, or it may explicitly declare itself to be a member of a different namespace. In either case, a unit declares its namespace membership in its unit header. For example, consider the following explicit namespace declaration:

```
unit MyCompany.MyWidgets.MyUnit;
```

First, notice that namespaces are separated by dots. Namespaces do not introduce new symbols for the identifiers between the dots; the dots are part of the unit name. The source file name for this example is MyCompany.MyWidgets.MyUnit.pas, and the compiled output file name is MyCompany.MyWidgets.MyUnit.dcuil.

Second, notice that the dots imply the conceptual nesting, or containment, of one namespace within another. The example above declares the unit `MyUnit` to be a member of the `MyWidgets` namespace, which itself is contained in the `MyCompany` namespace. Again, it should be noted that this containment is for documentation purposes only.

A project default namespace declares a namespace for all of the units in the project. Consider the following declarations:

```
Program MyCompany.Programs.MyProgram;
Library MyCompany.Libs.MyLibrary;
Package MyCompany.Packages.MyPackage;
```

These statements establish the project default namespace for the program, library, and package, respectively. The namespace is determined by removing the rightmost identifier (and dot) from the declaration.

A unit that omits an explicit namespace is called a *generic unit*. A generic unit automatically becomes a member of the project default namespace. Given the preceding program declaration, the following unit declaration would cause the compiler to treat `MyUnit` as a member of the `MyCompany.Programs` namespace.

```
unit MyUnit;
```

The project default namespace does not affect the name of the Delphi source file for a generic unit. In the preceding example, the Delphi source file name would be MyUnit.pas. The compiler does however prefix the dcuil file name with the project default namespace. The resulting dcuil file in the current example would be MyCompany.Programs.MyUnit.dcuil.

Namespace strings are not case-sensitive. The compiler considers two namespaces that differ only in case to be equivalent. However, the compiler does preserve the case of a namespace, and will use the preserved casing in output file names, error messages, and RTTI unit identifiers. RTTI for class and type names will include the full namespace specification.

Searching Namespaces

A unit must declare the other units on which it depends. As with the Win32 platform, the compiler must search these units for identifiers. For units in explicit namespaces the search scope is already known, but for generic units, the compiler must establish a namespace search scope.

Consider the following unit and uses declarations:

```
unit MyCompany.Programs.Units.MyUnit1;
uses MyCompany.Libs.Unit2, Unit3, Unit4;
```

These declarations establish `MyUnit1` as a member of the `MyCompany.Programs.Units` namespace. `MyUnit1` depends on three other units: `MyCompany.Libs.Unit2`, and the generic units, `Unit3`, and `Unit4`. The compiler can resolve identifier names in `Unit2`, since the uses clause specified the fully qualified unit name. To resolve identifier names in `Unit3` and `Unit4`, the compiler must establish a namespace search order.

Namespace search order

Search locations can come from three possible sources: Compiler options, the project default namespace, and finally, the current unit's namespace.

A project file (program, library or package) may optionally specify a list of namespaces to be searched when resolving generic unit names. The namespaces clause must appear in the project file, immediately after the program, library, or package declaration and before any other clause or block type. The namespaces clause is a list of namespace identifiers, separated by commas. A semicolon must terminate the list of namespaces.

The compiler resolves identifier names in the following order:

1. The current unit namespace (if any)
2. The project default namespace (if any)
3. Namespaces specified by compiler options.

A namespace search example

The following example project and unit files use the namespace clause to establish a namespace search list:

```
// Project file declarations...
program MyCompany.Programs.MyProgram;
namespaces MyCompany.Libs.UIWidgets, MyCompany.Libs.Network;
```

```
// Unit source file declaration...
unit MyCompany.Programs.Units.MyUnit1;
```

Given this program example, the compiler would search namespaces in the following order:

1. `MyCompany.Programs.Units`
2. `MyCompany.Programs`
3. `MyCompany.Libs.Network`
4. `MyCompany.Libs.UIWidgets`
5. Namespaces specified by compiler options.

Note that if the current unit is generic (i.e. it does not have an explicit namespace declaration in its unit statement), then resolution begins with the project default namespace.

Using Namespaces

Delphi's uses clause brings a module into the context of the current unit. The uses clause must either refer to a module by its fully qualified name (i.e. including the full namespace specification), or by its generic name, thereby relying on the namespace resolution mechanisms to locate the unit.

Fully qualified unit names

The following example demonstrates the uses clause with namespaces:

```
unit MyCompany.Libs.MyUnit1
uses MyCompany.Libs.Unit2,  // Fully qualified name.
     UnitX;                 // Generic name.
```

Once a module has been brought into context, source code can refer to identifiers within that module either by the unqualified name, or by the fully qualified name (if necessary, to disambiguate identifiers with the same name in different units). The following `writeln` statements are equivalent:

```
uses MyCompany.Libs.Unit2;

begin
   writeln(MyCompany.Libs.Unit2.SomeString);
   writeln(SomeString);
end.
```

A fully qualified identifier must include the full namespace specification. In the preceding example, it would be an error to refer to `SomeString` using only a portion of the namespace:

```
writeln(Unit2.SomeString);       // ERROR!
writeln(Libs.Unit2.SomeString);  // ERROR!
writeln(MyCompany.Libs.Unit2.SomeString);      // Correct.
writeln(SomeString);                           // Correct.
```

It is also an error to refer to only a portion of a namespace in the uses clause. There is no mechanism to import all units and symbols in a namespace. The following code does not import all units and symbols in the `MyCompany` namespace:

```
uses MyCompany;   // ERROR!
```

This restriction also applies to the with-do statement. The following will produce a compiler error:

```
with MyCompany.Libs do    // ERROR!
```

Namespaces and .NET Metadata

The Delphi for .NET compiler does not emit the entire dotted unit name into the assembly. Instead, the only leftmost portion - everything up to the last dot in the name is emitted. For example:

```

unit MyCompany.MyClasses.MyUnit
```

The compiler will emit the namespace `MyCompany.MyClasses` into the assembly metadata. This makes it easier for other .NET languages to call into Delphi assemblies.

This difference in namespace metadata is visible only to external consumers of the assembly. The Delphi code within the assembly still treats the entire dotted name as the fully qualified name.

Multi-unit Namespaces

Multiple units can be grouped together into one namespace using an extension of the in clause in the project source file. The file name string can list multiple unit source files in a semicolon-delimited list.

```

uses
  MyProgram.MyNamespace in 'filepath/unit1.pas;otherpath/unit2.pas';
```

In this example, the namespace `MyProgram.MyNamespace` logically contains all the interface symbols from `unit1` and `unit2`.

Symbol names in a namespace must be unique, across all units in the namespace. In the example above, if `unit1` and `unit2` both define a global interface symbol named `mySymbol`, the compiler will report an error in the uses clause.

You use a multi-unit namespace by specifying the namespace in the uses clause of a source file. Source code that uses a multi-unit namespace implicitly uses all the units listed as being members of that namespace.

All the units defined by the current project as members of that namespace will be available to the compiler for symbol lookup while compiling that source file.

The individual units aggregated in a namespace are not available to source code unless the individual units are explicitly used in the file's uses clause. In other words, if a source file uses only the namespace, then fully qualified identifier expressions referring to a symbol in a unit in that namespace must use the namespace name, not the actual unit that defines that symbol.

A uses clause may refer to a namespace as well as individual units within that namespace. In this case, a fully qualified expression referring to a symbol from a specific unit listed in the uses clause may be referred to using the actual unit name or the namespace for the qualifier. The two forms of reference are identical and refer to the same symbol.

Note: Since unit identifiers are not emitted to .NET assembly metadata, explicitly using the unit in the uses clause will only work when you are compiling from source or dcu files. If the namespace units are compiled into an assembly and the assembly is referenced by the project instead of the individual units, then the source code that explicitly refers to a unit in the namespace will fail because that unit name does not exist in the metadata of the assembly.

Fundamental Syntactic Elements

This topic introduces the Delphi language character set, and describes the syntax for declaring:

- Identifiers
- Numbers
- Character strings
- Labels
- Source code comments

The Delphi Character Set

The Delphi Language uses the Unicode character set, including alphabetic and alphanumeric Unicode characters and the underscore. It is not case-sensitive. The space character and the ASCII control characters (ASCII 0 through 31 including ASCII 13, the return or end-of-line character) are called *blanks*.

Fundamental syntactic elements, called *tokens*, combine to form expressions, declarations, and statements. A *statement* describes an algorithmic action that can be executed within a program. An *expression* is a syntactic unit that occurs within a statement and denotes a value. A *declaration* defines an identifier (such as the name of a function or variable) that can be used in expressions and statements, and, where appropriate, allocates memory for the identifier.

The Delphi Character Set and Basic Syntax

On the simplest level, a program is a sequence of tokens delimited by separators. A token is the smallest meaningful unit of text in a program. A separator is either a blank or a comment. Strictly speaking, it is not always necessary to place a separator between two tokens; for example, the code fragment

```
Size:=20;Price:=10;
```

is perfectly legal. Convention and readability, however, dictate that we write this as

```
Size := 20;
Price := 10;
```

Tokens are categorized as special symbols, identifiers, reserved words, directives, numerals, labels, and character strings. A separator can be part of a token only if the token is a character string. Adjacent identifiers, reserved words, numerals, and labels must have one or more separators between them.

Special Symbols

Special symbols are non-alphanumeric characters, or pairs of such characters, that have fixed meanings. The following single characters are special symbols:

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

The following character pairs are also special symbols:

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

The following table shows equivalent symbols:

Special symbol	Equivalent symbols
[	(.
]	.)
{	(*
}	*)

The left bracket **[** is equivalent to the character pair of left parenthesis and period **(.**

The right bracket **]** is equivalent to the character pair of period and right parenthesis **.)**

The left brace **{** is equivalent to the character pair of left parenthesis and asterisk **(***.

The right brace **}** is equivalent to the character pair of right parenthesis and asterisk ***)**

Note: **%**, **?**, \, **!**, **"** (double quotation marks), **_** (underscore), **|** (pipe), and **~** (tilde) are not special characters.

Identifiers

Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages. An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with an alphabetic character or an underscore (_) and cannot contain spaces; alphanumeric characters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers.

Note: The .NET SDK recommends against using leading underscores in identifiers, as this pattern is reserved for system use.

Since the Delphi Language is case-insensitive, an identifier like `CalculateValue` could be written in any of these ways:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

Since unit names correspond to file names, inconsistencies in case can sometimes affect compilation. For more information, see the topic, *Unit References and the Uses Clause*.

Qualified Identifiers

When you use an identifier that has been declared in more than one place, it is sometimes necessary to qualify the identifier. The syntax for a qualified identifier is

identifier1.identifier2

where *identifier1* qualifies *identifier2*. For example, if two units each declare a variable called `CurrentValue`, you can specify that you want to access the `CurrentValue` in `Unit2` by writing

```
Unit2.CurrentValue
```

Qualifiers can be iterated. For example,

```
Form1.Button1.Click
```

calls the `Click` method in `Button1` of `Form1`.

If you don't qualify an identifier, its interpretation is determined by the rules of scope described in Blocks and scope.

Extended Identifiers

When programming with Delphi for .NET, you might encounter CLR identifiers (e.g. types, or methods in a class) having the same name as a Delphi language keyword. For example, a class might have a method called `begin`. Another example is the CLR class called Type, in the System namespace. Type is a Delphi language keyword, and cannot be used for an identifier name.

If you qualify the identifier with its full namespace specification, then there is no problem. For example, to use the Type class, you must use its fully qualified name:

```
var
            TMyType : System.Type; // Using fully qualified namespace
                                   // avoides ambiguity with Delphi language keyword.
```

As a shorter alternative, Delphi for .NET introduces the ampersand (**&**) operator to resolve ambiguities between CLR identifiers and Delphi language keywords. If you encounter a method or type that is the same name as a Delphi keyword, you can omit the namespace specification if you prefix the identifier name with an ampersand. For example, the following code uses the ampersand to disambiguate the CLR Type class from the Delphi keyword type

```
var
           TMyType : &Type; // Prefix with '&' is ok.
```

Reserved Words

The following reserved words cannot be redefined or used as identifiers.

Reserved Words

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

In addition to the words above, private, protected, public, published, and automated act as reserved words within class type declarations, but are otherwise treated as directives. The words at and on also have special meanings, and should be treated as reserved words.

Directives

Directives are words that are sensitive in specific locations within source code. Directives have special meanings in the Delphi language, but, unlike reserved words, appear only in contexts where user-defined identifiers cannot occur. Hence -- although it is inadvisable to do so -- you can define an identifier that looks exactly like a directive.

Directives

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

Numerals

Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the **+** or **-** operator to indicate sign. Values default to positive (so that, for example, 67258 is equivalent to +67258) and must be within the range of the largest predefined real or integer type.

Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character E or e occurs within a real, it means "times ten to the power of". For example, 7E2 means 7 * 10^2, and 12.25e+6 and 12.25e6 both mean 12.25 * 10^6.

The dollar-sign prefix indicates a hexadecimal numeral, for example, $8F. Hexadecimal numbers without a preced ing **-** unary operator are taken to be positive values. During an assignment, if a hexadecimal value lies outside the range of the receiving type an error is raised, except in the case of the Integer (32-bit integer) where a warning is raised. In this case, values exceeding the positive range for Integer are taken to be negative numbers in a manner consistent with 2's complement integer representation.

For more information about real and integer types, see Data Types, Variables, and Constants. For information about the data types of numerals, see True constants.

Labels

A label is a standard Delphi language identifier with the exception that, unlike other identifiers, labels can start with a digit. Numeric labels can include no more than ten digits - that is, a numeral between 0 and 9999999999.

Labels are used in goto statements. For more information about goto statements and labels, see Goto statements.

Character Strings

A character string, also called a string literal or string constant, consists of a quoted string, a control string, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a null string. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe. For example,

```
'BORLAND' { BORLAND }
'You''ll see' { You'll see }
''''          { ' }
''            { null string }
' '           { a space }
```

A control string is a sequence of one or more control characters, each of which consists of the **#** symbol followed by an unsigned integer constant from 0 to 255 (decimal or hexadecimal) and denotes the corresponding ASCII character. The control string

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

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1'#13#10'Line 2'
```

to put a carriage-returnline-feed between 'Line 1' and 'Line 2'. However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the **+** operator or simply combine them into a single quoted string.)

A character string's length is the number of characters in the string. A character string of any length is compatible with any string type and with the PChar type. A character string of length 1 is compatible with any character type, and, when extended syntax is enabled (with compiler directive `{$X+}`), a nonempty character string of length n is compatible with zero-based arrays and packed arrays of n characters. For more information, see Datatypes, Variables, and Constants.

Comments and Compiler Directives

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives.

There are several ways to construct comments:

```
{ Text between a left brace and a right brace constitutes a comment. }
          (* Text between a left-parenthesis-plus-asterisk and an asterisk-plus-right-
parenthesis is also a comment *)
          // Any text between a double-slash and the end of the line constitutes a comment.
```

Comments that are alike cannot be nested. For instance, `{{}}` will not work, but `(*{}*)` will. This is useful for commenting out sections of code that also contain comments.

A comment that contains a dollar sign (**$**) immediately after the opening **{** or **(*** is a compiler directive. For example,

```
{$WARNINGS OFF}
```

tells the compiler not to generate warning messages.

Declarations and Statements

This topic describes the syntax of Delphi declarations and statements.

Aside from the uses clause (and reserved words like implementation that demarcate parts of a unit), a program consists entirely of *declarations* and *statements*, which are organized into *blocks*.

This topic covers the following items:

- Declarations
- Simple statements such as assignment
- Structured statements such as conditional tests (e.g., `if-then`, and case), iteration (e.g., `for`, and `while`).

Declarations

The names of variables, constants, types, fields, properties, procedures, functions, programs, units, libraries, and packages are called *identifiers*. (Numeric constants like 26057 are not identifiers.) Identifiers must be declared before you can use them; the only exceptions are a few predefined types, routines, and constants that the compiler understands automatically, the variable `Result` when it occurs inside a function block, and the variable `Self` when it occurs inside a method implementation.

A declaration defines an identifier and, where appropriate, allocates memory for it. For example,

```
var Size: Extended;
```

declares a variable called `Size` that holds an Extended (real) value, while

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

declares a function called `DoThis` that takes two strings as arguments and returns an integer. Each declaration ends with a semicolon. When you declare several variables, constants, types, or labels at the same time, you need only write the appropriate reserved word once:

```
var
   Size: Extended;
   Quantity: Integer;
   Description: string
```

The syntax and placement of a declaration depend on the kind of identifier you are defining. In general, declarations can occur only at the beginning of a block or at the beginning of the interface or implementation section of a unit (after the uses clause). Specific conventions for declaring variables, constants, types, functions, and so forth are explained in the documentation for those topics.

Hinting Directives

The 'hint' directives platform, deprecated, and library may be appended to any declaration. These directives will produce warnings at compile time. Hint directives can be applied to type declarations, variable declarations, class and structure declarations, field declarations within classes or records, procedure, function and method declarations, and unit declarations.

When a hint directive appears in a unit declaration, it means that the hint applies to everything in the unit. For example, the Windows 3.1 style OleAuto.pas unit on Windows is completely deprecated. Any reference to that unit or any symbol in that unit will produce a deprecation message.

The platform hinting directive on a symbol or unit indicates that it may not exist or that the implementation may vary considerably on different platforms. The library hinting directive on a symbol or unit indicates that the code may not exist or the implementation may vary considerably on different library architectures.

The platform and library directives do not specify which platform or library. If your goal is writing platform-independent code, you do not need to know which platform a symbol is specific to; it is sufficient that the symbol be marked as specific to *some* platform to let you know it may cause problems for your goal of portability.

In the case of a procedure or function declaration, the hint directive should be separated from the rest of the declaration with a semicolon. Examples:

```
procedure SomeOldRoutine; stdcall deprecated;

var
   VersionNumber: Real library;

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

When source code is compiled in the `{$HINTS ON} {$WARNINGS ON}` state, each reference to an identifier declared with one of these directives generates an appropriate hint or warning. Use platform to mark items that are specific to a particular operating environment (such as Windows or .NET), deprecated to indicate that an item is obsolete or supported only for backward compatibility, and library to flag dependencies on a particular library or component framework.

The Delphi 2005 compiler also recognizes the hinting directive experimental. You can use this directive to designate units which are in an unstable, development state. The compiler will emit a warning when it builds an application that uses the unit.

Statements

Statements define algorithmic actions within a program. Simple statements like assignments and procedure calls can combine to form loops, conditional statements, and other structured statements.

Multiple statements within a block, and in the initialization or finalization section of a unit, are separated by semicolons.

Simple Statements

A simple statement doesn't contain any other statements. Simple statements include assignments, calls to procedures and functions, and goto jumps.

Assignment Statements

An assignment statement has the form

variable := expression

where *variable* is any variable reference, including a variable, variable typecast, dereferenced pointer, or component of a structured variable. The *expression* is any assignment-compatible expression (within a function block, variable can be replaced with the name of the function being defined. See Procedures and functions). The **:=** symbol is sometimes called the assignment operator.

An assignment statement replaces the current value of variable with the value of expression. For example,

```
I := 3;
```

assigns the value 3 to the variable `I`. The variable reference on the left side of the assignment can appear in the expression on the right. For example,

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

increments the value of `I`. Other assignment statements include

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

Procedure and Function Calls

A procedure call consists of the name of a procedure (with or without qualifiers), followed by a parameter list (if required). Examples include

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

With extended syntax enabled (`{$X+}`), function calls, like calls to procedures, can be treated as statements in their own right:

```
MyFunction(X);
```

When you use a function call in this way, its return value is discarded.

For more information about procedures and functions, see Procedures and functions.

Goto Statements

A goto statement, which has the form

```
goto label
```

transfers program execution to the statement marked by the specified label. To mark a statement, you must first declare the label. Then precede the statement you want to mark with the label and a colon:

label: statement

Declare labels like this:

`label` *label;*

You can declare several labels at once:

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

A label can be any valid identifier or any numeral between 0 and 9999.

The label declaration, marked statement, and goto statement must belong to the same block. (See Blocks and Scope, below.) Hence it is not possible to jump into or out of a procedure or function. Do not mark more than one statement in a block with the same label.

For example,

```
label StartHere;
    ...
StartHere: Beep;
goto StartHere;
```

creates an infinite loop that calls the `Beep` procedure repeatedly.

Additionally, it is not possible to jump into or out of a try-finally or try-except statement.

The goto statement is generally discouraged in structured programming. It is, however, sometimes used as a way of exiting from nested loops, as in the following example.

```
procedure FindFirstAnswer;
  var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { some condition holds on X, Y, and Z } then
          goto FoundAnAnswer;

        ... { Code to execute if no answer is found }
        Exit;

FoundAnAnswer:
        ... { Code to execute when an answer is found }
end;
```

Notice that we are using goto to jump out of a nested loop. Never jump into a loop or other structured statement, since this can have unpredictable effects.

Structured Statements

Structured statements are built from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

- A compound or with statement simply executes a sequence of constituent statements.
- A conditional statement that is an if or case statement executes at most one of its constituents, depending on specified criteria.
- Loop statements including repeat, while, and for loops execute a sequence of constituent statements repeatedly.

- A special group of statements including raise, try...except, and try...finally constructions create and handle exceptions. For information about exception generation and handling, see Exceptions.

Compound Statements

A compound statement is a sequence of other (simple or structured) statements to be executed in the order in which they are written. The compound statement is bracketed by the reserved words begin and end, and its constituent statements are separated by semicolons. For example:

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

The last semicolon before end is optional. So we could have written this as

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

Compound statements are essential in contexts where Delphi syntax requires a single statement. In addition to program, function, and procedure blocks, they occur within other structured statements, such as conditionals or loops. For example:

```
begin
   I := SomeConstant;
   while I > 0 do
    begin
       ...
       I := I - 1;
    end;
end;
```

You can write a compound statement that contains only a single constituent statement; like parentheses in a complex term, begin and end sometimes serve to disambiguate and to improve readability. You can also use an empty compound statement to create a block that does nothing:

```
begin
end;
```

With Statements

A with statement is a shorthand for referencing the fields of a record or the fields, properties, and methods of an object. The syntax of a with statement is

with*obj*do*statement*

or

with*obj1, ..., objn*do*statement*

where *obj* is an expression yielding a reference to a record, object instance, class instance, interface or class type (metaclass) instance, and statement is any simple or structured statement. Within the *statement*, you can refer to fields, properties, and methods of *obj* using their identifiers alone, that is, without qualifiers.

For example, given the declarations

```
type
  TDate = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;

var
  OrderDate: TDate;
```

you could write the following with statement.

```
with OrderDate do
 if Month = 12 then
   begin
     Month := 1;
     Year := Year + 1;
   end
 else
   Month := Month + 1;
```

you could write the following with statement.

```
if OrderDate.Month = 12 then
  begin
   OrderDate.Month := 1;
   OrderDate.Year := OrderDate.Year + 1;
  end
else
  OrderDate.Month := OrderDate.Month + 1;
```

If the interpretation of *obj* involves indexing arrays or dereferencing pointers, these actions are performed once, before statement is executed. This makes with statements efficient as well as concise. It also means that assignments to a variable within statement cannot affect the interpretation of *obj* during the current execution of the with statement.

Each variable reference or method name in a with statement is interpreted, if possible, as a member of the specified object or record. If there is another variable or method of the same name that you want to access from the with statement, you need to prepend it with a qualifier, as in the following example.

```
with OrderDate do
  begin
    Year := Unit1.Year;
       ...
  end;
```

When multiple objects or records appear after with, the entire statement is treated like a series of nested with statements. Thus

with*obj1, obj2, ..., objn*do*statement*

is equivalent to

```
with obj1 do
  with obj2 do
    ...
    with objn do
      // statement
```

In this case, each variable reference or method name in statement is interpreted, if possible, as a member of *objn*; otherwise it is interpreted, if possible, as a member of *objn1*; and so forth. The same rule applies to interpreting the *objs* themselves, so that, for instance, if *objn* is a member of both *obj1* and *obj2*, it is interpreted as *obj2.objn*.

If Statements

There are two forms of if statement: if...then and the if...then...else. The syntax of an if...then statement is

if*expression*then*statement*

where *expression* returns a Boolean value. If expression is True, then *statement* is executed; otherwise it is not. For example,

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

The syntax of an if...then...else statement is

if*expression*then*statement1*else*statement2*

where *expression* returns a Boolean value. If expression is True, then *statement1* is executed; otherwise *statement2* is executed. For example,

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

The then and else clauses contain one statement each, but it can be a structured statement. For example,

```
if J <> o then
  begin
    Result := I / J;
    Count := Count + 1;
  end
else if Count = Last then
        Done := True
else
  Exit;
```

Notice that there is never a semicolon between the then clause and the word else. You can place a semicolon after an entire if statement to separate it from the next statement in its block, but the then and else clauses require nothing

more than a space or carriage return between them. Placing a semicolon immediately before else (in an if statement) is a common programming error.

A special difficulty arises in connection with nested if statements. The problem arises because some if statements have else clauses while others do not, but the syntax for the two kinds of statement is otherwise the same. In a series of nested conditionals where there are fewer else clauses than if statements, it may not seem clear which else clauses are bound to which ifs. Consider a statement of the form

if*expression1*thenif*expression2*then*statement1*else*statement2*;

There would appear to be two ways to parse this:

if*expression1* then [if*expression2*then*statement1*else*statement2*];

if*expression1*then [if*expression2*then*statement1*] else*statement2*;

The compiler always parses in the first way. That is, in real code, the statement

```
if ... { expression1} then
  if ... {expression2} then
    ... {statement1}
  else
    ... {statement2}
```

is equivalent to

```
if ... {expression1} then
  begin
    if ... {expression2} then
      ... {statement1}
    else
      ... {statement2}
end;
```

The rule is that nested conditionals are parsed starting from the innermost conditional, with each else bound to the nearest available if on its left. To force the compiler to read our example in the second way, you would have to write it explicitly as

```
if ... {expression1} then
  begin
   if ... {expression2} then
     ... {statement1}
   end
end
else
   ... {statement2};
```

Case Statements

The case statement may provide a readable alternative to deeply nested if conditionals. A case statement has the form

```
case selectorExpression of
   caseList1: statement1;
```

```
     ...
   caseListn: statementn;
end
```

where *selectorExpression* is any expression of an ordinal type (string types are invalid) and each *caseList* is one of the following:

- A numeral, declared constant, or other expression that the compiler can evaluate without executing your program. It must be of an ordinal type compatible with *selectorExpression*. Thus `7`, True, `4 + 5 * 3`, 'A', and `Integer('A')` can all be used as *caseLists*, but variables and most function calls cannot. (A few built-in functions like `Hi` and `Lo` can occur in a *caseList*. See Constant expressions.)
- A subrange having the form *First..Last*, where *First* and *Last* both satisfy the criterion above and *First* is less than or equal to *Last*.
- A list having the form *item1, ..., itemn*, where each *item* satisfies one of the criteria above.

Each value represented by a *caseList* must be unique in the case statement; subranges and lists cannot overlap. A case statement can have a final else clause:

```
case selectorExpression of
   caseList1: statement1;
      ...
   caselistn: statementn;
   else
      statements;
end
```

where *statements* is a semicolon-delimited sequence of statements. When a case statement is executed, at most one of *statement1 ... statementn* is executed. Whichever *caseList* has a value equal to that of *selectorExpression* determines the statement to be used. If none of the *caseLists* has the same value as *selectorExpression*, then the statements in the else clause (if there is one) are executed.

The case statement

```
case I of
   1..5: Caption := 'Low';
   6..9: Caption := 'High';
   0, 10..99: Caption := 'Out of range';
   else
     Caption := '';
end
```

is equivalent to the nested conditional

```
if I in [1..5] then
   Caption := 'Low';
else if I in [6..10] then
        Caption := 'High';
     else if (I = 0) or (I in [10..99]) then
            Caption := 'Out of range'
          else
            Caption := '';
```

Other examples of case statements

```
case MyColor of
   Red: X := 1;
   Green: X := 2;
   Blue: X = 3;
   Yellow, Orange, Black: X := 0;
end;

case Selection of
   Done: Form1.Close;
   Compute: calculateTotal(UnitCost, Quantity);
   else
      Beep;
end;
```

Control Loops

Loops allow you to execute a sequence of statements repeatedly, using a control condition or variable to determine when the execution stops. Delphi has three kinds of control loop: repeat statements, while statements, and for statements.

You can use the standard `Break` and `Continue` procedures to control the flow of a repeat, while, or for statement. `Break` terminates the statement in which it occurs, while `Continue` begins executing the next iteration of the sequence.

Repeat Statements

The syntax of a **repeat** statement is

repeat*statement1; ...; statementn;*until*expression*

where *expression* returns a Boolean value. (The last semicolon before until is optional.) The repeat statement executes its sequence of constituent statements continually, testing *expression* after each iteration. When *expression* returns True, the repeat statement terminates. The sequence is always executed at least once because *expression* is not evaluated until after the first iteration.

Examples of repeat statements include

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

repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

While Statements

A while statement is similar to a repeat statement, except that the control condition is evaluated before the first execution of the statement sequence. Hence, if the condition is false, the statement sequence is never executed.

The syntax of a while statement is

while*expression*do*statement*

where *expression* returns a Boolean value and *statement* can be a compound statement. The while statement executes its constituent *statement* repeatedly, testing *expression* before each iteration. As long as *expression* returns True, execution continues.

Examples of while statements include

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

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

  while not Eof(InputFile) do
   begin
      Readln(InputFile, Line);
      Process(Line);
   end;
```

For Statements

A for statement, unlike a repeat or while statement, requires you to specify explicitly the number of iterations you want the loop to go through. The syntax of a for statement is

for*counter* := *initialValue*to*finalValue*do*statement*

or

for*counter* := *initialValue*downto*finalValue*do*statement*

where

- *counter* is a local variable (declared in the block containing the for statement) of ordinal type, without any qualifiers.
- *initialValue* and *finalValue* are expressions that are assignment-compatible with counter.
- *statement* is a simple or structured statement that does not change the value of counter.

The for statement assigns the value of *initialValue* to *counter*, then executes statement repeatedly, incrementing or decrementing *counter* after each iteration. (The for...to syntax increments *counter*, while the for...downto syntax decrements it.) When *counter* returns the same value as *finalValue*, *statement* is executed once more and the for statement terminates. In other words, *statement* is executed once for every value in the range from *initialValue* to *finalValue*. If *initialValue* is equal to *finalValue*, *statement* is executed exactly once. If *initialValue* is greater than *finalValue* in a for...to statement, or less than *finalValue* in a for...downto statement, then *statement* is never executed. After the for statement terminates (provided this was not forced by a `Break` or an `Exit` procedure), the value of *counter* is undefined.

For purposes of controlling execution of the loop, the expressions *initialValue* and *finalValue* are evaluated only once, before the loop begins. Hence the for...to statement is almost, but not quite, equivalent to this while construction:

```
begin
   counter := initialValue;
   while counter <= finalValue do
    begin
      ... {statement};
```

```
      counter := Succ(counter);
    end;
end
```

The difference between this construction and the for...to statement is that the while loop reevaluates *finalValue* before each iteration. This can result in noticeably slower performance if *finalValue* is a complex expression, and it also means that changes to the value of *finalValue* within *statement* can affect execution of the loop.

Examples of for statements:

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

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

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

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

Iteration Over Containers Using For statements

Both Delphi for .NET and for Win32 support `for-element-in-collection` style iteration over containers. The following container iteration patterns are recognized by the compiler:

- `for Element in ArrayExpr do Stmt;`
- `for Element in StringExpr do Stmt;`
- `for Element in SetExpr do Stmt;`
- `for Element in CollectionExpr do Stmt;`

The type of the iteration variable `Element` must match the type held in the container. With each iteration of the loop, the iteration variable holds the current collection member. As with regular for-loops, the iteration variable must be declared within the same block as the for statement. The iteration variable cannot be modified within the loop. This includes assignment, and passing the variable to a var parameter of a procedure. Doing so will result in a compile-time error.

Array expressions may be single or multidimensional, fixed length, or dynamic arrays. The array is traversed in increasing order, starting at the lowest array bound and ending at the array size minus one. The code below shows an example of traversing single, multi-dimensional, and dynamic arrays:

```
type
  TIntArray        = array[0..9] of Integer;
  TGenericIntArray = array of Integer;

var
  IArray1: array[0..9] of Integer   = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  IArray2: array[1..10] of Integer  = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
  IArray3: array[1..2] of TIntArray = ((11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
                                       (21, 22, 23, 24, 25, 26, 27, 28, 29, 30));
  MultiDimTemp: TIntArray;
  IDynArray:    TGenericIntArray;

  I: Integer;

begin

  for I in IArray1 do
     begin
         // Do something with I...
     end;

  // Indexing begins at lower array bound of 1.
  for I in IArray2 do
     begin
         // Do something with I...
     end;

  // Iterating a multi-dimensional array
  for MultiDimTemp in IArray3 do   // Indexing from 1..2
    for I in MultiDimTemp do       // Indexing from 0..9
      begin
          // Do something with I...
      end;

  // Iterating over a dynamic array
  IDynArray := IArray1;
  for I in IDynArray do
     begin
        // Do something with I...
     end;
```

The following code example demonstrates iteration over string expressions:

```
var
  C: Char;
  S1, S2: String;
  Counter: Integer;

  OS1, OS2: ShortString;
  AC: AnsiChar;

begin

  S1 := 'Now is the time for all good men to come to the aid of their country.';
  S2 := '';

  for C in S1 do
    S2 := S2 + C;

  if S1 = S2 then
    WriteLn('SUCCESS #1');
  else
    WriteLn('FAIL #1');

  OS1 := 'When in the course of human events it becomes necessary to dissolve...';
  OS2 := '';
```

```
  for AC in OS1 do
    OS2 := OS2 + AC;

  if OS1 = OS2 then
    WriteLn('SUCCESS #2');
  else
    WriteLn('FAIL #2');

end.
```

The following code example demonstrates iteration over set expressions:

```
type

  TMyThing = (one, two, three);
  TMySet   = set of TMyThing;
  TCharSet = set of Char;

var
  MySet:   TMySet;
  MyThing: TMyThing;

  CharSet: TCharSet;
  {$IF DEFINED(CLR)}
  C: AnsiChar;
  {$ELSE}
  C: Char;
  {$IFEND}

begin

  MySet := [one, two, three];
  for MyThing in MySet do
   begin
    // Do something with MyThing...
   end;


  CharSet := [#0..#255];
  for C in CharSet do
    begin
      // Do something with C...
    end;

end.
```

To use the for-in loop construct on a class, the class must implement a prescribed collection pattern. A type that implements the collection pattern must have the following attributes:

- The class must contain a public instance method called `GetEnumerator()`. The `GetEnumerator()` method must return a class, interface, or record type.
- The class, interface, or record returned by `GetEnumerator()` must contain a public instance method called `MoveNext()`. The `MoveNext()` method must return a Boolean.
- The class, interface, or record returned by `GetEnumerator()` must contain a public instance, read-only property called `Current`. The type of the `Current` property must be the type contained in the collection.

If the enumerator type returned by `GetEnumerator()` implements the IDisposable interface, the compiler will call the type's Dispose method when the loop terminates.

The following code demonstrates iterating over an enumerable container in Delphi.

```
type
  TMyIntArray = array of Integer;

  TMyEnumerator = class
    Values: TMyIntArray;
    Index:  Integer;
  public
    constructor Create;
    function GetCurrent: Integer;
    function MoveNext:   Boolean;
    property Current:    Integer read GetCurrent;
  end;

  TMyContainer  = class
  public
   function GetEnumerator: TMyEnumerator;
  end;

constructor TMyEnumerator.Create;
begin
  inherited Create;
  Values := TMyIntArray.Create(100, 200, 300);
  Index := -1;
end;

function TMyEnumerator.MoveNext: Boolean;
begin
  if Index < High(Values) then
    begin
      Inc(Index);
      Result := True;
    end
  else
    Result := False;
end;

function TMyEnumerator.GetCurrent: Integer;
begin
  Result := Values[Index];
end;

function TMyContainer.GetEnumerator: TMyThing;
begin
  Result := TMyEnumerator.Create;
end;

var
  MyContainer: TMyContainer;
  I: Integer;

  Counter: Integer;

begin
  MyContainer := TMyContainer.Create;
```

```
  Counter := 0;
  for I in MyContainer do
    Inc(Counter, I);

  WriteLn('Counter = ', Counter);
end.
```

The following classes and their descendents support the for-in syntax:

- TList
- TCollection
- TStrings
- TInterfaceList
- TComponent
- TMenuItem
- TCustomActionList
- TFields
- TListItems
- TTreeNodes
- TToolBar

Blocks and Scope

Declarations and statements are organized into *blocks*, which define local namespaces (or *scopes*) for labels and identifiers. Blocks allow a single identifier, such as a variable name, to have different meanings in different parts of a program. Each block is part of the declaration of a program, function, or procedure; each program, function, or procedure declaration has one block.

Blocks

A block consists of a series of declarations followed by a compound statement. All declarations must occur together at the beginning of the block. So the form of a block is

```
{declarations}
begin
  {statements}
end
```

The *declarations* section can include, in any order, declarations for variables, constants (including resource strings), types, procedures, functions, and labels. In a program block, the *declarations* section can also include one or more exports clauses (see Libraries and packages).

For example, in a function declaration like

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
```

```
    ...
end;
```

the first line of the declaration is the function heading and all of the succeeding lines make up the block. `Ch`, `L`, `Source`, and `Dest` are local variables; their declarations apply only to the `UpperCase` function block and override, in this block only, any declarations of the same identifiers that may occur in the program block or in the interface or implementation section of a unit.

Scope

An identifier, such as a variable or function name, can be used only within the scope of its declaration. The location of a declaration determines its scope. An identifier declared within the declaration of a program, function, or procedure has a scope limited to the block in which it is declared. An identifier declared in the interface section of a unit has a scope that includes any other units or programs that use the unit where the declaration occurs. Identifiers with narrower scope, especially identifiers declared in functions and procedures, are sometimes called local, while identifiers with wider scope are called global.

The rules that determine identifier scope are summarized below.

If the identifier is declared in ...	its scope extends ...
the declaration section of a program, function, or procedure	from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.
the interface section of a unit	from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit. (See Programs and Units.)
the implementation section of a unit, but not within the block of any function or procedure	from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit, including the initialization and finalization sections, if present.
the definition of a record type (that is, the identifier is the name of a field in the record)	from the point of its declaration to the end of the record-type definition. (See Records.)
the definition of a class (that is, the identifier is the name of a data field property or method in the class)	from the point of its declaration to the end of the class-type definition, and also includes descendants of the class and the blocks of all methods in the class and its descendants. (See Classes and Objects.)

Naming Conflicts

When one block encloses another, the former is called the outer block and the latter the inner block. If an identifier declared in an outer block is redeclared in an inner block, the inner declaration takes precedence over the outer one and determines the meaning of the identifier for the duration of the inner block. For example, if you declare a variable called `MaxValue` in the interface section of a unit, and then declare another variable with the same name in a function declaration within that unit, any unqualified occurrences of `MaxValue` in the function block are governed by the second, local declaration. Similarly, a function declared within another function creates a new, inner scope in which identifiers used by the outer function can be redeclared locally.

The use of multiple units further complicates the definition of scope. Each unit listed in a uses clause imposes a new scope that encloses the remaining units used and the program or unit containing the uses clause. The first unit in a uses clause represents the outermost scope and each succeeding unit represents a new scope inside the previous one. If two or more units declare the same identifier in their interface sections, an unqualified reference to the identifier selects the declaration in the innermost scope, that is, in the unit where the reference itself occurs, or, if that unit doesn't declare the identifier, in the last unit in the `uses` clause that does declare the identifier.

The System and SysInit units are used automatically by every program or unit. The declarations in System, along with the predefined types, routines, and constants that the compiler understands automatically, always have the outermost scope.

You can override these rules of scope and bypass an inner declaration by using a qualified identifier (see Qualified Identifiers) or a with statement (see With Statements, above).

Expressions

This topic describes syntax rules of forming Delphi expressions.

The following items are covered in this topic:

- Valid Delphi Expressions
- Operators
- Function calls
- Set constructors
- Indexes
- Typecasts

Expressions

An expression is a construction that returns a value. The following table shows examples of Delphi expressions:

`X`	variable
`@X`	address of the variable X
`15`	integer constant
`InterestRate`	variable
`Calc(X, Y)`	function call
`X * Y`	product of X and Y
`Z / (1 - Z)`	quotient of Z and (1 - Z)
`X = 1.5`	Boolean
`C in Range1`	Boolean
`not Done`	negation of a Boolean
`['a', 'b', 'c']`	set
`Char(48)`	value typecast

The simplest expressions are variables and constants (described in Data types, variables, and constants). More complex expressions are built from simpler ones using operators, function calls, set constructors, indexes, and typecasts.

Operators

Operators behave like predefined functions that are part of the the Delphi language. For example, the expression `(X + Y)` is built from the variables `X` and `Y`, called operands, with the **+** operator; when `X` and `Y` represent integers or reals, `(X + Y)` returns their sum. Operators include **@**, **not**, **^**, *****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**, **+**, **-**, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **in**, and **is**.

The operators **@**, **not**, and **^** are unary (taking one operand). All other operators are binary (taking two operands), except that **+** and **-** can function as either a unary or binary operator. A unary operator always precedes its operand (for example, `-B`), except for **^**, which follows its operand (for example, `P^`). A binary operator is placed between its operands (for example, `A = 7`).

Some operators behave differently depending on the type of data passed to them. For example, not performs bitwise negation on an integer operand and logical negation on a Boolean operand. Such operators appear below under multiple categories.

Except for **^**, is, and in, all operators can take operands of type Variant.

The sections that follow assume some familiarity with Delphi data types.

For information about operator precedence in complex expressions, see Operator Precedence Rules, later in this topic.

Arithmetic Operators

Arithmetic operators, which take real or integer operands, include **+**, **-**, *****, **/**, div, and mod.

Binary Arithmetic Operators

Operator	Operation	Operand Types	Result Type	Example
+	addition	integer, real	integer, real	`X + Y`
-	subtraction	integer, real	integer, real	`Result - 1`
*	multiplication	integer, real	integer, real	`P * InterestRate`
/	real division	integer, real	real	`X / 2`
div	integer division	integer	integer	`Total div UnitSize`
mod	remainder	integer	integer	`Y mod 6`

Unary arithmetic operators

Operator	Operation	Operand Type	Result Type	Example
+	sign identity	integer, real	integer, real	`+7`
-	sign negation	integer, real	integer, real	`-X`

The following rules apply to arithmetic operators.

- The value of `x / y` is of type Extended, regardless of the types of `x` and `y`. For other arithmetic operators, the result is of type Extended whenever at least one operand is a real; otherwise, the result is of type Int64 when at least one operand is of type Int64; otherwise, the result is of type Integer. If an operand's type is a subrange of an integer type, it is treated as if it were of the integer type.
- The value of `x div y` is the value of `x / y` rounded in the direction of zero to the nearest integer.
- The mod operator returns the remainder obtained by dividing its operands. In other words, `x mod y = x (x div y) * y`.
- A runtime error occurs when y is zero in an expression of the form `x / y`, `x div y`, or `x mod y`.

Boolean Operators

The Boolean operators not, and, or, and xor take operands of any Boolean type and return a value of type Boolean.

Boolean Operators

Operator	Operation	Operand Types	Result Type	Example
not	negation	Boolean	Boolean	`not (C in MySet)`
and	conjunction	Boolean	Boolean	`Done and (Total > 0)`
or	disjunction	Boolean	Boolean	`A or B`

xor	exclusive disjunction	Boolean	Boolean	`A xor B`

These operations are governed by standard rules of Boolean logic. For example, an expression of the form `x and y` is True if and only if both `x` and `y` are True.

Complete Versus Short-Circuit Boolean Evaluation

The compiler supports two modes of evaluation for the and and or operators: complete evaluation and short-circuit (partial) evaluation. Complete evaluation means that each conjunct or disjunct is evaluated, even when the result of the entire expression is already determined. Short-circuit evaluation means strict left-to-right evaluation that stops as soon as the result of the entire expression is determined. For example, if the expression `A and B` is evaluated under short-circuit mode when `A` is False, the compiler won't evaluate `B`; it knows that the entire expression is False as soon as it evaluates `A`.

Short-circuit evaluation is usually preferable because it guarantees minimum execution time and, in most cases, minimum code size. Complete evaluation is sometimes convenient when one operand is a function with side effects that alter the execution of the program.

Short-circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string `S`, up to the first comma.

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

In the case where `S` has no commas, the last iteration increments `I` to a value which is greater than the length of `S`. When the while condition is next tested, complete evaluation results in an attempt to read `S[I]`, which could cause a runtime error. Under short-circuit evaluation, in contrast, the second part of the while condition `(S[I] <> ',')` is not evaluated after the first part fails.

Use the `$B` compiler directive to control evaluation mode. The default state is `{$B}`, which enables short-circuit evaluation. To enable complete evaluation locally, add the `{$B+}` directive to your code. You can also switch to complete evaluation on a project-wide basis by selecting **Complete Boolean Evaluation** in the **Compiler Options** dialog (all source units will need to be recompiled).

Note: If either operand involves a Variant, the compiler always performs complete evaluation (even in the `{$B}` state).

Logical (Bitwise) Operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in `X` (in binary) is `001101` and the value stored in `Y` is `100001`, the statement

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

assigns the value `101101` to `Z`.

Logical (Bitwise) Operators

Operator	Operation	Operand Types	Result Type	Example
not	bitwise negation	integer	integer	`not X`
and	bitwise and	integer	integer	`X and Y`

or	bitwise or	integer	integer	`X or Y`
xor	bitwise xor	integer	integer	`X xor Y`
shl	bitwise shift left	integer	integer	`X shl 2`
shr	bitwise shift right	integer	integer	`Y shr I`

The following rules apply to bitwise operators.

- The result of a not operation is of the same type as the operand.
- If the operands of an and, or, or xor operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations `x shl y` and `x shr y` shift the value of `x` to the left or right by `y` bits, which (if `x` is an unsigned integer) is equivalent to multiplying or dividing `x` by `2^y`; the result is of the same type as `x`. For example, if `N` stores the value `01101` (decimal 13), then `N sh 1` returns `11010` (decimal 26). Note that the value of `y` is interpreted modulo the size of the type of `x`. Thus for example, if `x` is an integer, `x shl 40` is interpreted as `x shl 8` because an integer is 32 bits and 40 mod 32 is 8.

String Operators

The relational operators **=**, **<>**, **<**, **>**, **<=**, and **>=** all take string operands (see Relational operators). The **+** operator concatenates two strings.

String Operators

Operator	Operation	Operand Types	Result Type	Example
+	concatenation	string, packed string, character	string	`S + '. '`

The following rules apply to string concatenation.

- The operands for **+** can be strings, packed strings (packed arrays of type Char), or characters. However, if one operand is of type WideChar, the other operand must be a long string (AnsiString or WideString).
- The result of a **+** operation is compatible with any string type. However, if the operands are both short strings or characters, and their combined length is greater than 255, the result is truncated to the first 255 characters.

Pointer Operators

The relational operators **<**, **>**, **<=**, and **>=** can take operands of type PChar and PWideChar (see Relational operators). The following operators also take pointers as operands. For more information about pointers, see Pointers and pointer types.

Character-pointer operators

Operator	Operation	Operand Types	Result Type	Example
+	pointer addition	character pointer, integer	character pointer	`P + I`
-	pointer subtraction	character pointer, integer	character pointer, integer	`P - Q`
^	pointer dereference	pointer	base type of pointer	`P^`
=	equality	pointer	Boolean	`P = Q`
<>	inequality	pointer	Boolean	`P <> Q`

The **^** operator dereferences a pointer. Its operand can be a pointer of any type except the generic Pointer, which must be typecast before dereferencing.

`P = Q` is True just in case `P` and `Q` point to the same address; otherwise, `P <> Q` is True.

You can use the **+** and **-** operators to increment and decrement the offset of a character pointer. You can also use **-** to calculate the difference between the offsets of two character pointers. The following rules apply.

- If `I` is an integer and `P` is a character pointer, then `P + I` adds `I` to the address given by `P`; that is, it returns a pointer to the address `I` characters after `P`. (The expression `I + P` is equivalent to `P + I`.) `P - I` subtracts `I` from the address given by `P`; that is, it returns a pointer to the address `I` characters before `P`. This is true for PChar pointers; for PWideChar pointers `P + I` adds `SizeOf(WideChar)` to `P`.
- If `P` and `Q` are both character pointers, then `P - Q` computes the difference between the address given by `P` (the higher address) and the address given by `Q` (the lower address); that is, it returns an integer denoting the number of characters between `P` and `Q`. `P + Q` is not defined.

Set Operators

The following operators take sets as operands.

Set Operators

Operator	Operation	Operand Types	Result Type	Example
+	union	set	set	`Set1 + Set2`
-	difference	set	set	`S - T`
*****	intersection	set	set	`S * T`
<=	subset	set	Boolean	`Q <= MySet`
>=	superset	set	Boolean	`S1 >= S2`
=	equality	set	Boolean	`S2 = MySet`
<>	inequality	set	Boolean	`MySet <> S1`
in	membership	ordinal, set	Boolean	`A in Set1`

The following rules apply to **+**, **-**, and *****.

- An ordinal `O` is in `X + Y` if and only if `O` is in `X` or `Y` (or both). `O` is in `X - Y` if and only if `O` is in `X` but not in `Y`. `O` is in `X * Y` if and only if `O` is in both `X` and `Y`.
- The result of a **+**, **-**, or ***** operation is of the type `set of A..B`, where `A` is the smallest ordinal value in the result set and `B` is the largest.

The following rules apply to **<=**, **>=**, **=**, **<>**, and **in**.

- `X <= Y` is True just in case every member of `X` is a member of `Y`; `Z >= W` is equivalent to `W <= Z`. `U = V` is True just in case `U` and `V` contain exactly the same members; otherwise, `U <> V` is True.
- For an ordinal `O` and a set `S`, `O in S` is True just in case `O` is a member of `S`.

Relational Operators

Relational operators are used to compare two operands. The operators **=**, **<>**, **<=**, and **>=** also apply to sets.

Relational Operators

Operator	Operation	Operand Types	Result Type	Example
=	equality	simple, class, class reference, interface, string, packed string	Boolean	`I = Max`

<>	inequality	simple, class, class reference, interface, string, packed string	Boolean	`X <> Y`
<	less-than	simple, string, packed string, PChar	Boolean	`X < Y`
>	greater-than	simple, string, packed string, PChar	Boolean	`Len > 0`
<=	less-than-or-equal-to	simple, string, packed string, PChar	Boolean	`Cnt <= I`
>=	greater-than-or-equal-to	simple, string, packed string, PChar	Boolean	`I >= 1`

For most simple types, comparison is straightforward. For example, `I = J` is True just in case `I` and `J` have the same value, and `I <> J` is True otherwise. The following rules apply to relational operators.

- Operands must be of compatible types, except that a real and an integer can be compared.
- Strings are compared according to the ordinal values that make up the characters that make up the string. Character types are treated as strings of length 1.
- Two packed strings must have the same number of components to be compared. When a packed string with `n` components is compared to a string, the packed string is treated as a string of length `n`.
- Use the operators **<**, **>**, **<=**, and **>=** to compare PChar (and PWideChar) operands only if the two pointers point within the same character array.
- The operators **=** and **<>** can take operands of class and class-reference types. With operands of a class type, **=** and **<>** are evaluated according the rules that apply to pointers: `C = D` is True just in case `C` and `D` point to the same instance object, and `C <> D` is True otherwise. With operands of a class-reference type, `C = D` is True just in case `C` and `D` denote the same class, and `C <> D` is True otherwise. This does not compare the data stored in the classes. For more information about classes, see Classes and objects.

Class Operators

The operators as and is take classes and instance objects as operands; as operates on interfaces as well. For more information, see Classes and objects and Object interfaces.

The relational operators **=** and **<>** also operate on classes.

The @ Operator

The **@** operator returns the address of a variable, or of a function, procedure, or method; that is, **@** constructs a pointer to its operand. For more information about pointers, see Pointers and pointer types. The following rules apply to **@**.

- If `X` is a variable, `@X` returns the address of `X`. (Special rules apply when `X` is a procedural variable; see Procedural types in statements and expressions.) The type of `@X` is Pointer if the default `{$T}` compiler directive is in effect. In the `{$T+}` state, `@X` is of type `^T`, where `T` is the type of `X` (this distinction is important for assignment compatibility, see Assignment-compatibility).
- If `F` is a routine (a function or procedure), `@F` returns `F`'s entry point. The type of `@F` is always Pointer.
- When **@** is applied to a method defined in a class, the method identifier must be qualified with the class name. For example,

```
@TMyClass.DoSomething
```

points to the `DoSomething` method of `TMyClass`. For more information about classes and methods, see Classes and objects.

Note: When using the **@** operator, it is not possible to take the address of an interface method as the address is not known at compile time and cannot be extracted at runtime.

Operator Precedence

In complex expressions, rules of precedence determine the order in which operations are performed.

Precedence of operators

Operators	Precedence
@, not	first (highest)
*, /, div, mod, and, shl, shr, as	second
+, **-**, or, xor	third
=, **<>**, **<**, **>**, **<=**, **>=**, in, is	fourth (lowest)

An operator with higher precedence is evaluated before an operator with lower precedence, while operators of equal precedence associate to the left. Hence the expression

```
X + Y * Z
```

multiplies `Y` times `Z`, then adds `X` to the result; ***** is performed first, because is has a higher precedence than **+**. But

```
X - Y + Z
```

first subtracts `Y` from `X`, then adds `Z` to the result; **-** and **+** have the same precedence, so the operation on the left is performed first.

You can use parentheses to override these precedence rules. An expression within parentheses is evaluated first, then treated as a single operand. For example,

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

multiplies `Z` times the sum of `X` and `Y`.

Parentheses are sometimes needed in situations where, at first glance, they seem not to be. For example, consider the expression

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

The intended interpretation of this is obviously

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

Without parentheses, however, the compiler follows operator precedence rules and reads it as

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

which results in a compilation error unless `Z` is Boolean.

Parentheses often make code easier to write and to read, even when they are, strictly speaking, superfluous. Thus the first example could be written as

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

Here the parentheses are unnecessary (to the compiler), but they spare both programmer and reader from having to think about operator precedence.

Function Calls

Because functions return a value, function calls are expressions. For example, if you've defined a function called `Calc` that takes two integer arguments and returns an integer, then the function call `Calc(24,47)` is an integer expression. If `I` and `J` are integer variables, then `I + Calc(J,8)` is also an integer expression. Examples of function calls include

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I,J);
```

For more information about functions, see Procedures and functions.

Set Constructors

A set constructor denotes a set-type value. For example,

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

denotes the set whose members are 5, 6, 7, and 8. The set constructor

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

could also denote the same set.

The syntax for a set constructor is

[item1, ..., itemn]

where each item is either an expression denoting an ordinal of the set's base type or a pair of such expressions with two dots (**..**) in between. When an item has the form `x..y`, it is shorthand for all the ordinals in the range from `x` to `y`, including `y`; but if `x` is greater than `y`, then `x..y`, the set `[x..y]`, denotes nothing and is the empty set. The set constructor `[ ]` denotes the empty set, while `[x]` denotes the set whose only member is the value of `x`.

Examples of set constructors:

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

For more information about sets, see Sets.

Indexes

Strings, arrays, array properties, and pointers to strings or arrays can be indexed. For example, if `FileName` is a string variable, the expression `FileName[3]` returns the third character in the string denoted by `FileName`, while `FileName[I + 1]` returns the character immediately after the one indexed by `I`. For information about strings, see String types. For information about arrays and array properties, see Arrays and Array properties.

Typecasts

It is sometimes useful to treat an expression as if it belonged to different type. A typecast allows you to do this by, in effect, temporarily changing an expression's type. For example, `Integer('A')` casts the character `A` as an integer.

The syntax for a typecast is

typeIdentifier(expression)

If the expression is a variable, the result is called a variable typecast; otherwise, the result is a value typecast. While their syntax is the same, different rules apply to the two kinds of typecast.

Value Typecasts

In a value typecast, the type identifier and the cast expression must both be ordinal or pointer types. Examples of value typecasts include

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

The resulting value is obtained by converting the expression in parentheses. This may involve truncation or extension if the size of the specified type differs from that of the expression. The expression's sign is always preserved.

The statement

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

assigns the value of `Integer('A')`, which is 65, to the variable `I`.

A value typecast cannot be followed by qualifiers and cannot appear on the left side of an assignment statement.

Variable Typecasts

You can cast any variable to any type, provided their sizes are the same and you do not mix integers with reals. (To convert numeric types, rely on standard functions like `Int` and `Trunc`.) Examples of variable typecasts include

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

Variable typecasts can appear on either side of an assignment statement. Thus

```
var MyChar: char;
  ...
  Shortint(MyChar) := 122;
```

assigns the character `z` (ASCII 122) to `MyChar`.

You can cast variables to a procedural type. For example, given the declarations

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

you can make the following assignments.

```
F := Func(P);      { Assign procedural value in P to F }
Func(P) := F;      { Assign procedural value in F to P }
@F := P;           { Assign pointer value in P to F }
P := @F;           { Assign pointer value in F to P }
N := F(N);         { Call function via F }
N := Func(P)(N);   { Call function via P }
```

Variable typecasts can also be followed by qualifiers, as illustrated in the following example.

```
type
  TByteRec = record
     Lo, Hi: Byte;
  end;

  TWordRec = record
     Low, High: Word;
  end;

var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;

begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $1234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

In this example, `TByteRec` is used to access the low- and high-order bytes of a word, and `TWordRec` to access the low- and high-order words of a long integer. You could call the predefined functions `Lo` and `Hi` for the same purpose, but a variable typecast has the advantage that it can be used on the left side of an assignment statement.

For information about typecasting pointers, see Pointers and pointer types. For information about casting class and interface types, see The as operator and Interface typecasts.

Data Types, Variables, and Constants

This section describes the fundamental data types of the Delphi language.

Data Types, Variables, and Constants

This topic presents a high-level overview of Delphi data types.

About Types

A type is essentially a name for a kind of data. When you declare a variable you must specify its type, which determines the set of values the variable can hold and the operations that can be performed on it. Every expression returns data of a particular type, as does every function. Most functions and procedures require parameters of specific types.

The Delphi language is a 'strongly typed' language, which means that it distinguishes a variety of data types and does not always allow you to substitute one type for another. This is usually beneficial because it lets the compiler treat data intelligently and validate your code more thoroughly, preventing hard-to-diagnose runtime errors. When you need greater flexibility, however, there are mechanisms to circumvent strong typing. These include typecasting, pointers, Variants, Variant parts in records, and absolute addressing of variables.

There are several ways to categorize Delphi data types:

- Some types are predefined (or built-in); the compiler recognizes these automatically, without the need for a declaration. Almost all of the types documented in this language reference are predefined. Other types are created by declaration; these include user-defined types and the types defined in the product libraries.
- Types can be classified as either fundamental or generic. The range and format of a fundamental type is the same in all implementations of the Delphi language, regardless of the underlying CPU and operating system. The range and format of a generic type is platform-specific and could vary across different implementations. Most predefined types are fundamental, but a handful of integer, character, string, and pointer types are generic. It's a good idea to use generic types when possible, since they provide optimal performance and portability. However, changes in storage format from one implementation of a generic type to the next could cause compatibility problems - for example, if you are streaming content to a file as raw, binary data, without type and versioning information.
- Types can be classified as simple, string, structured, pointer, procedural, or variant. In addition, type identifiers themselves can be regarded as belonging to a special 'type' because they can be passed as parameters to certain functions (such as High, Low, and SizeOf).

The outline below shows the taxonomy of Delphi data types.

```
simple
            ordinal
              integer
              character
              Boolean
              enumerated
              subrange
             real
           string
           structured
            set
            array
            record
            file
            class
            class reference
            interface
          pointer
          procedural
          Variant
          type identifier
```

The standard function `SizeOf` operates on all variables and type identifiers. It returns an integer representing the amount of memory (in bytes) required to store data of the specified type. For example, `SizeOf(Longint)` returns 4, since a Longint variable uses four bytes of memory.

Type declarations are illustrated in the topics that follow. For general information about type declarations, see Declaring types.

Simple Types

Simple types - which include ordinal types and real types - define ordered sets of values.

The ordinal types covered in this topic are:

- Integer types
- Character types
- Boolean types
- Enumerated types
- Real (floating point) types

Ordinal Types

Ordinal types include integer, character, Boolean, enumerated, and subrange types. An ordinal type defines an ordered set of values in which each value except the first has a unique predecessor and each value except the last has a unique successor. Further, each value has an ordinality which determines the ordering of the type. In most cases, if a value has ordinality *n*, its predecessor has ordinality *n-1* and its successor has ordinality *n+1*.

- For integer types, the ordinality of a value is the value itself.
- Subrange types maintain the ordinalities of their base types.
- For other ordinal types, by default the first value has ordinality 0, the next value has ordinality 1, and so forth. The declaration of an enumerated type can explicitly override this default.

Several predefined functions operate on ordinal values and type identifiers. The most important of them are summarized below.

Function	Parameter	Return value	Remarks
`Ord`	ordinal expression	ordinality of expression's value	Does not take Int64 arguments.
`Pred`	ordinal expression	predecessor of expression's value	
`Succ`	ordinal expression	successor of expression's value	
`High`	ordinal type identifier or variable of ordinal type	highest value in type	Also operates on short-string types and arrays.
`Low`	ordinal type identifier or variable of ordinal type	lowest value in type	Also operates on short-string types and arrays.

For example, `High(Byte)` returns 255 because the highest value of type Byte is 255, and `Succ(2)` returns 3 because 3 is the successor of 2.

The standard procedures `Inc` and `Dec` increment and decrement the value of an ordinal variable. For example, `Inc(I)` is equivalent to `I := Succ(I)` and, if `I` is an integer variable, to `I := I + 1`.

Integer Types

An integer type represents a subset of the whole numbers. The generic integer types are Integer and Cardinal; use these whenever possible, since they result in the best performance for the underlying CPU and operating system. The table below gives their ranges and storage formats for the Delphi compiler.

Generic integer types

Type	Range	Format	.NET Type Mapping
Integer	-2147483648..2147483647	signed 32-bit	Int32
Cardinal	0..4294967295	unsigned 32-bit	UInt32

Fundamental integer types include Shortint, Smallint, Longint, Int64, Byte, Word, and Longword.

Fundamental integer types

Type	Range	Format	.NET Type Mapping
Shortint	-128..127	signed 8-bit	SByte
Smallint	-32768..32767	signed 16-bit	Int16
Longint	-2147483648..2147483647	signed 32-bit	Int32
Int64	-2^63..2^631	signed 64-bit	Int64
Byte	0..255	unsigned 8-bit	Byte
Word	0..65535	unsigned 16-bit	UInt16
Longword	0..4294967295	unsigned 32-bit	UInt32

In general, arithmetic operations on integers return a value of type Integer, which is equivalent to the 32-bit Longint. Operations return a value of type Int64 only when performed on one or more Int64 operand. Hence the following code produces incorrect results.

```
var
  I: Integer;
  J: Int64;
   ...

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

To get an Int64 return value in this situation, cast `I` as Int64:

```
...
J := Int64(I) + 1;
```

For more information, see Arithmetic operators.

Note: Some standard routines that take integer arguments truncate Int64 values to 32 bits. However, the `High`, `Low`, `Succ`, `Pred`, `Inc`, `Dec`, `IntToStr`, and `IntToHex` routines fully support Int64 arguments. Also, the `Round`, `Trunc`, `StrToInt64`, and `StrToInt64Def` functions return Int64 values. A few routines cannot take Int64 values at all.

When you increment the last value or decrement the first value of an integer type, the result wraps around the beginning or end of the range. For example, the Shortint type has the range 128..127; hence, after execution of the code

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

the value of `I` is 128. If compiler range-checking is enabled, however, this code generates a runtime error.

Character Types

The fundamental character types are AnsiChar and WideChar. AnsiChar values are byte-sized (8-bit) characters ordered according to the locale character set which is possibly multibyte. AnsiChar was originally modeled after the ANSI character set (thus its name) but has now been broadened to refer to the current locale character set.

WideChar characters use more than one byte to represent every character. In the current implementations, WideChar is word-sized (16-bit) characters ordered according to the Unicode character set (note that it could be longer in future implementations). The first 256 Unicode characters correspond to the ANSI characters.

The generic character type is Char, which is equivalent to AnsiChar on Win32, and to Char on the .NET platform. Because the implementation of Char is subject to change, it's a good idea to use the standard function `SizeOf` rather than a hard-coded constant when writing programs that may need to handle characters of different sizes.

Note: The WideChar type also maps to Char on the .NET platform.

A string constant of length 1, such as 'A', can denote a character value. The predefined function `Chr` returns the character value for any integer in the range of AnsiChar or WideChar; for example, `Chr(65)` returns the letter A.

Character values, like integers, wrap around when decremented or incremented past the beginning or end of their range (unless range-checking is enabled). For example, after execution of the code

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

`Letter` has the value A (ASCII 65).

Boolean Types

The four predefined Boolean types are Boolean, ByteBool, WordBool, and LongBool. Boolean is the preferred type. The others exist to provide compatibility with other languages and operating system libraries.

A Boolean variable occupies one byte of memory, a ByteBool variable also occupies one byte, a WordBool variable occupies two bytes (one word), and a LongBool variable occupies four bytes (two words).

Boolean values are denoted by the predefined constants True and False. The following relationships hold.

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

A value of type ByteBool, LongBool, or WordBool is considered True when its ordinality is nonzero. If such a value appears in a context where a Boolean is expected, the compiler automatically converts any value of nonzero ordinality to True.

The previous remarks refer to the ordinality of Boolean values, not to the values themselves. In Delphi, Boolean expressions cannot be equated with integers or reals. Hence, if X is an integer variable, the statement

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

generates a compilation error. Casting the variable to a Boolean type is unreliable, but each of the following alternatives will work.

```
if X <> 0 then ...; { use longer expression that returns Boolean value }

var OK: Boolean;
 ...
if X <> 0 then OK := True;
if OK then ...;
```

Enumerated Types

An enumerated type defines an ordered set of values by simply listing identifiers that denote these values. The values have no inherent meaning. To declare an enumerated type, use the syntax

type*typeName* = (*val1* , ..., *valn*)

where *typeName* and each *val* are valid identifiers. For example, the declaration

```
type Suit = (Club, Diamond, Heart, Spade);
```

defines an enumerated type called `Suit` whose possible values are `Club`, `Diamond`, `Heart`, and `Spade`, where `Ord(Club)` returns 0, `Ord(Diamond)` returns 1, and so forth.

When you declare an enumerated type, you are declaring each val to be a constant of type *typeName*. If the *val* identifiers are used for another purpose within the same scope, naming conflicts occur. For example, suppose you declare the type

```
type TSound = (Click, Clack, Clock)
```

Unfortunately, `Click` is also the name of a method defined for TControl and all of the objects in VCL that descend from it. So if you're writing an application and you create an event handler like

```
procedure TForm1.DBGridEnter(Sender: TObject);
var Thing: TSound;
begin
   ...
   Thing := Click;
end;
```

you'll get a compilation error; the compiler interprets `Click` within the scope of the procedure as a reference to `TForm`'s `Click` method. You can work around this by qualifying the identifier; thus, if `TSound` is declared in `MyUnit`, you would use

```
Thing := MyUnit.Click;
```

A better solution, however, is to choose constant names that are not likely to conflict with other identifiers. Examples:

```
type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe)
```

You can use the *(val1, ..., valn)* construction directly in variable declarations, as if it were a type name:

```
var MyCard: (Club, Diamond, Heart, Spade);
```

But if you declare `MyCard` this way, you can't declare another variable within the same scope using these constant identifiers. Thus

```
var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);
```

generates a compilation error. But

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

compiles cleanly, as does

```
type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;
```

Enumerated Types with Explicitly Assigned Ordinality

By default, the ordinalities of enumerated values start from 0 and follow the sequence in which their identifiers are listed in the type declaration. You can override this by explicitly assigning ordinalities to some or all of the values in the declaration. To assign an ordinality to a value, follow its identifier with = *constantExpression*, where *constantExpression* is a constant expression that evaluates to an integer. For example,

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

defines a type called `Size` whose possible values include `Small`, `Medium`, and `Large`, where `Ord(Small)` returns 5, `Ord(Medium)` returns 10, and `Ord(Large)` returns 15.

An enumerated type is, in effect, a subrange whose lowest and highest values correspond to the lowest and highest ordinalities of the constants in the declaration. In the previous example, the `Size` type has 11 possible values whose ordinalities range from 5 to 15. (Hence the type `array[Size] of Char` represents an array of 11 characters.) Only three of these values have names, but the others are accessible through typecasts and through routines such as `Pred`, `Succ`, `Inc`, and `Dec`. In the following example, "anonymous" values in the range of `Size` are assigned to the variable `X`.

```
var X: Size;

  X := Small;    // Ord(X) = 5
  Y := Size(6);  // Ord(X) = 6
  Inc(X);        // Ord(X) = 7
```

Any value that isn't explicitly assigned an ordinality has ordinality one greater than that of the previous value in the list. If the first value isn't assigned an ordinality, its ordinality is 0. Hence, given the declaration

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

`SomeEnum` has only two possible values: `Ord(e1)` returns 0, `Ord(e2)` returns 1, and `Ord(e3)` also returns 1; because `e2` and `e3` have the same ordinality, they represent the same value.

Enumerated constants without a specific value have RTTI:

```
type SomeEnum = (e1, e2, e3);
```

whereas enumerated constants with a specific value, such as the following, do not have RTTI:

```
type SomeEnum = (e1 = 1, e2 = 2, e3 = 3);
```

Subrange Types

A subrange type represents a subset of the values in another ordinal type (called the base type). Any construction of the form *Low..High*, where *Low* and *High* are constant expressions of the same ordinal type and *Low* is less than *High*, identifies a subrange type that includes all values between *Low* and *High*. For example, if you declare the enumerated type

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

you can then define a subrange type like

```
type TMyColors = Green..White;
```

Here `TMyColors` includes the values `Green`,`Yellow`, `Orange`, `Purple`, and `White`.

You can use numeric constants and characters (string constants of length 1) to define subrange types:

```
type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';
```

When you use numeric or character constants to define a subrange, the base type is the smallest integer or character type that contains the specified range.

The *LowerBound..UpperBound* construction itself functions as a type name, so you can use it directly in variable declarations. For example,

```
var SomeNum: 1..500;
```

declares an integer variable whose value can be anywhere in the range from 1 to 500.

The ordinality of each value in a subrange is preserved from the base type. (In the first example, if `Color` is a variable that holds the value `Green`, `Ord(Color)` returns 2 regardless of whether `Color` is of type `TColors` or `TMyColors`.) Values do not wrap around the beginning or end of a subrange, even if the base is an integer or character type; incrementing or decrementing past the boundary of a subrange simply converts the value to the base type. Hence, while

```
type  Percentile = 0..99;
var  I: Percentile;
  ...
  I := 100;
```

produces an error,

```
...
  I := 99;
  Inc(I);
```

assigns the value 100 to `I` (unless compiler range-checking is enabled).

The use of constant expressions in subrange definitions introduces a syntactic difficulty. In any type declaration, when the first meaningful character after **=** is a left parenthesis, the compiler assumes that an enumerated type is being defined. Hence the code

```
const
  X = 50;
  Y = 10;

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

produces an error. Work around this problem by rewriting the type declaration to avoid the leading parenthesis:

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

Real Types

A real type defines a set of numbers that can be represented with floating-point notation. The table below gives the ranges and storage formats for the fundamental real types on the Win32 platform.

Fundamental Win32 real types

Type	Range	Significant digits	Size in bytes
Real48	-2.9 x 10^39 .. 1.7 x 10^38	1112	6
Single	-1.5 x 10^45 .. 3.4 x 10^38	78	4
Double	-5.0 x 10^324 .. 1.7 x 10^308	1516	8

Extended	-3.6 x 10^4951 .. 1.1 x 10^4932	1920	10
Comp	-2^63+1 .. 2^63 1	1920	8
Currency	-922337203685477.5808.. 922337203685477.5807	1920	8

The following table shows how the fundamental real types map to .NET framework types.

Fundamental .NET real type mappings

Type	.NET Mapping
Real48	Deprecated
Single	Single
Double	Double
Extended	Double
Comp	Deprecated
Currency	Re-implemented as a value type using the Decimal type from the .NET Framework

The generic type Real, in its current implementation, is equivalent to Double (which maps to Double on .NET).

Generic real types

Type	Range	Significant digits	Size in bytes
Real	-5.0 x 10^324 .. 1.7 x 10^308	1516	8

Note: The six-byte Real48 type was called Real in earlier versions of Object Pascal. If you are recompiling code that uses the older, six-byte Real type in Delphi, you may want to change it to Real48. You can also use the `{$REALCOMPATIBILITY ON}` compiler directive to turn Real back into the six-byte type.

The following remarks apply to fundamental real types.

- Real48 is maintained for backward compatibility. Since its storage format is not native to the Intel processor architecture, it results in slower performance than other floating-point types. The Real48 type has been deprecated on the .NET platform.
- Extended offers greater precision than other real types but is less portable. Be careful using Extended if you are creating data files to share across platforms.
- The Comp (computational) type is native to the Intel processor architecture and represents a 64-bit integer. It is classified as a real, however, because it does not behave like an ordinal type. (For example, you cannot increment or decrement a Comp value.) Comp is maintained for backward compatibility only. Use the Int64 type for better performance.
- Currency is a fixed-point data type that minimizes rounding errors in monetary calculations. On the Win32 platform, it is stored as a scaled 64-bit integer with the four least significant digits implicitly representing decimal places. When mixed with other real types in assignments and expressions, Currency values are automatically divided or multiplied by 10000.

String Types

This topic describes the string data types available in the Delphi language. The following types are covered:

- Short strings.
- Long strings.
- Wide (Unicode) strings.

About String Types

A string represents a sequence of characters. Delphi supports the following predefined string types.

String types

Type	Maximum length	Memory required	Used for
ShortString	255 characters	2 to 256 bytes	backward compatibility
AnsiString	~2^31 characters	4 bytes to 2GB	8-bit (ANSI) characters, DBCS ANSI, MBCS ANSI, etc.
WideString	~2^30 characters	4 bytes to 2GB	Unicode characters; multi-user servers and multi-language applications

On the Win32 platform, AnsiString, sometimes called the long string, is the preferred type for most purposes. WideString is the preferred string type on the .NET platform.

String types can be mixed in assignments and expressions; the compiler automatically performs required conversions. But strings passed by reference to a function or procedure (as var and out parameters) must be of the appropriate type. Strings can be explicitly cast to a different string type.

The reserved word string functions like a generic type identifier. For example,

```
var S: string;
```

creates a variable `S` that holds a string. On the Win32 platform, the compiler interprets string (when it appears without a bracketed number after it) as AnsiString. On the .NET platform, the string type maps to the String class. You can use single byte character strings on the .NET platform, but you must explicitly declare them to be of type AnsiString.

On the Win32 platform, you can use the `{$H-}` directive to turn string into ShortString. The `{$H-}` directive is deprecated on the .NET platform.

The standard function `Length` returns the number of characters in a string. The `SetLength` procedure adjusts the length of a string.

Comparison of strings is defined by the ordering of the characters in corresponding positions. Between strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value. For example, 'AB' is greater than 'A'; that is, 'AB' > 'A' returns True. Zero-length strings hold the lowest values.

You can index a string variable just as you would an array. If `S` is a string variable and `i` an integer expression, `S[i]` represents the *ith* character - or, strictly speaking, the *ith* byte in `S`. For a ShortString or AnsiString, `S[i]` is of type AnsiChar; for a WideString, `S[i]` is of type WideChar. For single-byte (Western) locales, `MyString[2] := 'A';` assigns the value `A` to the second character of `MyString`. The following code uses the standard `AnsiUpperCase` function to convert `MyString` to uppercase.

```
var I: Integer;
begin
   I := Length(MyString);
```

```
    while I > 0 do
     begin
        MyString[I] := AnsiUpperCase(MyString[I]);
        I := I - 1;
     end;
end;
```

Be careful indexing strings in this way, since overwriting the end of a string can cause access violations. Also, avoid passing long-string indexes as var parameters, because this results in inefficient code.

You can assign the value of a string constant - or any other expression that returns a string - to a variable. The length of the string changes dynamically when the assignment is made. Examples:

```
MyString := 'Hello world!';
MyString := 'Hello' + 'world';
MyString := MyString + '!';
MyString := ' '; { space }
MyString := '';  { empty string }
```

Short Strings

A ShortString is 0 to 255 characters long. While the length of a ShortString can change dynamically, its memory is a statically allocated 256 bytes; the first byte stores the length of the string, and the remaining 255 bytes are available for characters. If `S` is a ShortString variable, `Ord(S[0])`, like `Length(S)`, returns the length of `S`; assigning a value to `S[0]`, like calling `SetLength`, changes the length of `S`. ShortString is maintained for backward compatibility only.

The Delphi language supports short-string types - in effect, subtypes of ShortString - whose maximum length is anywhere from 0 to 255 characters. These are denoted by a bracketed numeral appended to the reserved word string. For example,

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

creates a variable called `MyString` whose maximum length is 100 characters. This is equivalent to the declarations

```
type CString = string[100];
var MyString: CString;
```

Variables declared in this way allocate only as much memory as the type requires - that is, the specified maximum length plus one byte. In our example, `MyString` uses 101 bytes, as compared to 256 bytes for a variable of the predefined ShortString type.

When you assign a value to a short-string variable, the string is truncated if it exceeds the maximum length for the type.

The standard functions High and Low operate on short-string type identifiers and variables. High returns the maximum length of the short-string type, while Low returns zero.

Long Strings

AnsiString, also called a long string, represents a dynamically allocated string whose maximum length is limited only by available memory.

A long-string variable is a pointer occupying four bytes of memory. When the variable is empty - that is, when it contains a zero-length stringthe pointer is nil and the string uses no additional storage. When the variable is

nonempty, it points a dynamically allocated block of memory that contains the string value. The eight bytes before the location contain a 32-bit length indicator and a 32-bit reference count. This memory is allocated on the heap, but its management is entirely automatic and requires no user code.

Because long-string variables are pointers, two or more of them can reference the same value without consuming additional memory. The compiler exploits this to conserve resources and execute assignments faster. Whenever a long-string variable is destroyed or assigned a new value, the reference count of the old string (the variable's previous value) is decremented and the reference count of the new value (if there is one) is incremented; if the reference count of a string reaches zero, its memory is deallocated. This process is called reference-counting. When indexing is used to change the value of a single character in a string, a copy of the string is made if - but only if - its reference count is greater than one. This is called copy-on-write semantics.

WideString

The WideString type represents a dynamically allocated string of 16-bit Unicode characters. In most respects it is similar to AnsiString. On Win32, WideString is compatible with the COM BSTR type.

Note: Under Win32, WideString values are not reference-counted.

The Win32 platform supports single-byte and multibyte character sets as well as Unicode. With a single-byte character set (SBCS), each byte in a string represents one character.

In a multibyte character set (MBCS), some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. The null value (#0) is always a single-byte character. Multibyte character sets - especially double-byte character sets (DBCS) - are widely used for Asian languages.

In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words. Unicode characters and strings are also called wide characters and wide character strings. The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2).

The Delphi language supports single-byte and multibyte characters and strings through the Char, PChar, AnsiChar, PAnsiChar, and AnsiString types. Indexing of multibyte strings is not reliable, since `S[i]` represents the *ith* byte (not necessarily the *ith* character) in `S`. However, the standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. (Names of multibyte functions usually start with `Ansi`-. For example, the multibyte version of `StrPos` is `AnsiStrPos`.) Multibyte character support is operating-system dependent and based on the current locale.

Delphi supports Unicode characters and strings through the WideChar, PWideChar, and WideString types.

Working with null-Terminated Strings

Many programming languages, including C and C++, lack a dedicated string data type. These languages, and environments that are built with them, rely on null-terminated strings. A null-terminated string is a zero-based array of characters that ends with NUL (#0); since the array has no length indicator, the first NUL character marks the end of the string. You can use Delphi constructions and special routines in the `SysUtils` unit (see Standard routines and I/O) to handle null-terminated strings when you need to share data with systems that use them.

For example, the following type declarations could be used to store null-terminated strings.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;
```

With extended syntax enabled (`{$X+}`), you can assign a string constant to a statically allocated zero-based character array. (Dynamic arrays won't work for this purpose.) If you initialize an array constant with a string that is shorter than the declared length of the array, the remaining characters are set to `#0`.

Using Pointers, Arrays, and String Constants

To manipulate null-terminated strings, it is often necessary to use pointers. (See Pointers and pointer types.) String constants are assignment-compatible with the PChar and PWideChar types, which represent pointers to null-terminated arrays of Char and WideChar values. For example,

```
var P: PChar;
  ...
P := 'Hello world!'
```

points `P` to an area of memory that contains a null-terminated copy of 'Hello world!' This is equivalent to

```
const TempString: array[0..12] of Char = 'Hello world!';
var P: PChar;
   ...
P := @TempString[0];
```

You can also pass string constants to any function that takes value or const parameters of type PChar or PWideChar - for example `StrUpper('Hello world!')`. As with assignments to a PChar, the compiler generates a null-terminated copy of the string and gives the function a pointer to that copy. Finally, you can initialize PChar or PWideChar constants with string literals, alone or in a structured type. Examples:

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = ('Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six',
'Seven', 'Eight', 'Nine');
```

Zero-based character arrays are compatible with PChar and PWideChar. When you use a character array in place of a pointer value, the compiler converts the array to a pointer constant whose value corresponds to the address of the first element of the array. For example,

```
var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;
```

This code calls `SomeProcedure` twice with the same value.

A character pointer can be indexed as if it were an array. In the previous example, `MyPointer[0]` returns `H`. The index specifies an offset added to the pointer before it is dereferenced. (For PWideChar variables, the index is automatically multiplied by two.) Thus, if `P` is a character pointer, `P[0]` is equivalent to `P^` and specifies the first character in the array, `P[1]` specifies the second character in the array, and so forth; `P[-1]` specifies the 'character' immediately to the left of `P[0]`. The compiler performs no range checking on these indexes.

The `StrUpper` function illustrates the use of pointer indexing to iterate through a null-terminated string:

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

Mixing Delphi Strings and Null-Terminated Strings

You can mix long strings (AnsiString values) and null-terminated strings (PChar values) in expressions and assignments, and you can pass PChar values to functions or procedures that take long-string parameters. The assignment `S := P`, where `S` is a string variable and `P` is a PChar expression, copies a null-terminated string into a long string.

In a binary operation, if one operand is a long string and the other a PChar, the PChar operand is converted to a long string.

You can cast a PChar value as a long string. This is useful when you want to perform a string operation on two PChar values. For example,

```
S := string(P1) + string(P2);
```

You can also cast a long string as a null-terminated string. The following rules apply.

- If `S` is a long-string expression, `PChar(S)` casts `S` as a null-terminated string; it returns a pointer to the first character in `S`. For example, if `Str1` and `Str2` are long strings, you could call the Win32 API `MessageBox` function like this: `MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);`

- You can also use `Pointer(S)` to cast a long string to an untyped pointer. But if `S` is empty, the typecast returns nil.
- `PChar(S)` always returns a pointer to a memory block; if `S` is empty, a pointer to `#0` is returned.
- When you cast a long-string variable to a pointer, the pointer remains valid until the variable is assigned a new value or goes out of scope. If you cast any other long-string expression to a pointer, the pointer is valid only within the statement where the typecast is performed.
- When you cast a long-string expression to a pointer, the pointer should usually be considered read-only. You can safely use the pointer to modify the long string only when all of the following conditions are satisfied.
- The expression cast is a long-string variable.
- The string is not empty.
- The string is unique - that is, has a reference count of one. To guarantee that the string is unique, call the SetLength, `SetString`, or `UniqueString` procedure.
- The string has not been modified since the typecast was made.
- The characters modified are all within the string. Be careful not to use an out-of-range index on the pointer.

The same rules apply when mixing WideString values with PWideChar values.

Structured Types

Instances of a structured type hold more than one value. Structured types include sets, arrays, records, and files as well as class, class-reference, and interface types. Except for sets, which hold ordinal values only, structured types can contain other structured types; a type can have unlimited levels of structuring.

By default, the values in a structured type are aligned on word or double-word boundaries for faster access. When you declare a structured type, you can include the reserved word packed to implement compressed data storage. For example, `type TNumbers = packed array [1..100] of Real;`

Using packed slows data access and, in the case of a character array, affects type compatibility (for more information, see Memory management).

This topic covers the following structured types:

- Sets
- Arrays, including static and dynamic arrays.
- Records
- File types

Sets

A set is a collection of values of the same ordinal type. The values have no inherent order, nor is it meaningful for a value to be included twice in a set.

The range of a set type is the power set of a specific ordinal type, called the base type; that is, the possible values of the set type are all the subsets of the base type, including the empty set. The base type can have no more than 256 possible values, and their ordinalities must fall between 0 and 255. Any construction of the form

`set of` *baseType*

where *baseType* is an appropriate ordinal type, identifies a set type.

Because of the size limitations for base types, set types are usually defined with subranges. For example, the declarations

```
type
 TSomeInts = 1..250;
 TIntSet = set of TSomeInts;
```

create a set type called `TIntSet` whose values are collections of integers in the range from 1 to 250. You could accomplish the same thing with

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

Given this declaration, you can create a sets like this:

```
var Set1, Set2: TIntSet;
    ...
    Set1 := [1, 3, 5, 7, 9];
    Set2 := [2, 4, 6, 8, 10]
```

You can also use the `set of ...` construction directly in variable declarations:

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

Other examples of set types include

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

The in operator tests set membership:

```
if 'a' in MySet then ... { do something } ;
```

Every set type can hold the empty set, denoted by [].

Arrays

An array represents an indexed collection of elements of the same type (called the base type). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once. Arrays can be allocated *statically* or *dynamically*.

Static Arrays

Static array types are denoted by constructions of the form

`array[` *indexType1*, ..., *indexTypen* `] of` *baseType*;

where each *indexType* is an ordinal type whose range does not exceed 2GB. Since the *indexTypes* index the array, the number of elements an array can hold is limited by the product of the sizes of the *indexTypes*. In practice, *indexTypes* are usually integer subranges.

In the simplest case of a one-dimensional array, there is only a single *indexType*. For example,

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

declares a variable called `MyArray` that holds an array of 100 character values. Given this declaration, `MyArray [3]` denotes the third character in `MyArray`. If you create a static array but don't assign values to all its elements, the unused elements are still allocated and contain random data; they are like uninitialized variables.

A multidimensional array is an array of arrays. For example,

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

is equivalent to

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

Whichever way `TMatrix` is declared, it represents an array of 500 real values. A variable `MyMatrix` of type `TMatrix` can be indexed like this: `MyMatrix[2,45];` or like this: `MyMatrix[2][45]`. Similarly,

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

is equivalent to

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

The standard functions `Low` and `High` operate on array type identifiers and variables. They return the low and high bounds of the array's first index type. The standard function `Length` returns the number of elements in the array's first dimension.

A one-dimensional, packed, static array of Char values is called a packed string. Packed-string types are compatible with string types and with other packed-string types that have the same number of elements. See Type compatibility and identity.

An array type of the form `array[0..x] of Char` is called a zero-based character array. Zero-based character arrays are used to store null-terminated strings and are compatible with PChar values. See Working with null-terminated strings.

Dynamic Arrays

Dynamic arrays do not have a fixed size or length. Instead, memory for a dynamic array is reallocated when you assign a value to the array or pass it to the `SetLength` procedure. Dynamic-array types are denoted by constructions of the form

`array of` *baseType*

For example,

```
var MyFlexibleArray: array of Real;
```

declares a one-dimensional dynamic array of reals. The declaration does not allocate memory for `MyFlexibleArray`. To create the array in memory, call `SetLength`. For example, given the previous declaration,

```
SetLength(MyFlexibleArray, 20);
```

allocates an array of 20 reals, indexed 0 to 19. Dynamic arrays are always integer-indexed, always starting from 0.

Dynamic-array variables are implicitly pointers and are managed by the same reference-counting technique used for long strings. To deallocate a dynamic array, assign nil to a variable that references the array or pass the variable to `Finalize`; either of these methods disposes of the array, provided there are no other references to it. Dynamic arrays are automatically released when their reference-count drops to zero. Dynamic arrays of length 0 have the value nil. Do not apply the dereference operator (**^**) to a dynamic-array variable or pass it to the `New` or `Dispose` procedure.

If `X` and `Y` are variables of the same dynamic-array type, `X := Y` points `X` to the same array as `Y`. (There is no need to allocate memory for `X` before performing this operation.) Unlike strings and static arrays, *copy-on-write* is not employed for dynamic arrays, so they are not automatically copied before they are written to. For example, after this code executes,

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

```
    B[0] := 2;
  end;
```

the value of `A[0]` is 2. (If `A` and `B` were static arrays, `A[0]` would still be 1.)

Assigning to a dynamic-array index (for example, `MyFlexibleArray[2] := 7`) does not reallocate the array. Out-of-range indexes are not reported at compile time.

In contrast, to make an independent copy of a dynamic array, you must use the global `Copy` function:

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

When dynamic-array variables are compared, their references are compared, not their array values. Thus, after execution of the code

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

`A = B` returns False but `A[0] = B[0]` returns True.

To truncate a dynamic array, pass it to `SetLength`, or pass it to `Copy` and assign the result back to the array variable. (The `SetLength` procedure is usually faster.) For example, if `A` is a dynamic array, `A := SetLength(A, 0, 20)` truncates all but the first 20 elements of `A`.

Once a dynamic array has been allocated, you can pass it to the standard functions `Length`, `High`, and `Low`. `Length` returns the number of elements in the array, `High` returns the array's highest index (that is, `Length - 1`), and `Low` returns 0. In the case of a zero-length array, `High` returns 1 (with the anomalous consequence that `High < Low`).

Note: In some function and procedure declarations, array parameters are represented as `array of` *baseType*, without any index types specified. For example,`function CheckStrings(A: array of string): Boolean;`

This indicates that the function operates on all arrays of the specified base type, regardless of their size, how they are indexed, or whether they are allocated statically or dynamically.

Multidimensional Dynamic Arrays

To declare multidimensional dynamic arrays, use iterated `array of ...` constructions. For example,

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

declares a two-dimensional array of strings. To instantiate this array, call `SetLength` with two integer arguments. For example, if `I` and `J` are integer-valued variables,

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

allocates an *I-by-J* array, and `Msgs[0,0]` denotes an element of that array.

You can create multidimensional dynamic arrays that are not rectangular. The first step is to call `SetLength`, passing it parameters for the first n dimensions of the array. For example,

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

allocates ten rows for `Ints` but no columns. Later, you can allocate the columns one at a time (giving them different lengths); for example

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

makes the third column of `Ints` five integers long. At this point (even if the other columns haven't been allocated) you can assign values to the third column - for example, `Ints[2,4] := 6`.

The following example uses dynamic arrays (and the IntToStr function declared in the `SysUtils` unit) to create a triangular matrix of strings.

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

Array Types and Assignments

Arrays are assignment-compatible only if they are of the same type. Because the Delphi language uses name-equivalence for types, the following code will not compile.

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

To make the assignment work, declare the variables as

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

or

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

Dynamically allocated multidimensional arrays (.NET)

On the .NET platform, multidimensional arrays can be dynamically allocated using the New standard function. Using the New syntax to allocate an array, the array declaration specifies the number of dimensions, but not their actual size. You then pass the element type, the actual array dimensions, or an array initializer list to the New function. The array declaration has the following syntax:

`array[, ..., ] of` *baseType*;

In the syntax, note that the number of dimensions are specified by using a comma as a placeholder; the actual size is not determined until runtime, when you call the New function. There are two forms of the New function: one takes the element type and the size of the array, and the other takes the element type and an array initializer list. The following code demonstrates both forms:

```
var
  a: array [,,] of integer;  // 3 dimensional array
  b: array [,] of integer;   // 2 dimensional array
  c: array [,] of TPoint;    // 2 dimensional array of TPoint

begin
   a := New(array[3,5,7] of integer);                 // New taking element type and size
of each dimension.
   b := New(array[,] of integer, ((1,2,3), (4,5,6))); // New taking the element type and
initializer list.
                                                      // New taking an initializer list of TPoint.
   c := New(array[,] of TPoint, (((X:1;Y:2), (X:3;Y:4)), ((X:5;Y:6), (X:7;Y:8))));
end.
```

You can allocate the array by passing variable or constant expressions to the New function:

```
var
   a:   array[,] of integer;
   r,c: Integer;

begin
   r := 4;
   c := 17;

   a := New(array [r,c] of integer);
```

You can also use the SetLength procedure to allocate the array, by passing the array expression, and the size of each dimension, for example:

```
var
  a: array[,] of integer;
  b: array[,,] of integer;

begin
     SetLength(a, 4,5);
     SetLength(b, 3,5,7);
end.
```

The Copy function can be used to make a copy of an entire array. You cannot use Copy to duplicate only a portion, however.

You cannot pass a dynamically allocated rectangular array to the Low or High functions. Attempting to do so will generate a compile-time error.

Records

A record (analogous to a structure in some languages) represents a heterogeneous set of elements. Each element is called a field; the declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```
type recordTypeName = record
       fieldList1: type1;
        ...
       fieldListn: typen;
     end
```

where *recordTypeName* is a valid identifier, each type denotes a type, and each fieldList is a valid identifier or a comma-delimited list of identifiers. The final semicolon is optional.

For example, the following declaration creates a record type called `TDateRec`.

```
type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;
```

Each `TDateRec` contains three fields: an integer value called `Year`, a value of an enumerated type called `Month`, and another integer between 1 and 31 called `Day`. The identifiers `Year`, `Month`, and `Day` are the field designators for `TDateRec`, and they behave like variables. The `TDateRec` type declaration, however, does not allocate any memory for the `Year`, `Month`, and `Day` fields; memory is allocated when you instantiate the record, like this:

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

This variable declaration creates two instances of `TDateRec`, called `Record1` and `Record2`.

You can access the fields of a record by qualifying the field designators with the record's name:

```
Record1.Year := 1904;
```

```
Record1.Month := Jun;
Record1.Day := 16;
```

Or use a with statement:

```
with Record1 do
 begin
    Year := 1904;
    Month := Jun;
    Day := 16;
 end;
```

You can now copy the values of `Record1`'s fields to `Record2`:

```
Record2 := Record1;
```

Because the scope of a field designator is limited to the record in which it occurs, you don't have to worry about naming conflicts between field designators and other variables.

Instead of defining record types, you can use the `record ...` construction directly in variable declarations:

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

However, a declaration like this largely defeats the purpose of records, which is to avoid repetitive coding of similar groups of variables. Moreover, separately declared records of this kind will not be assignment-compatible, even if their structures are identical.

Variant Parts in Records

A record type can have a variant part, which looks like a case statement. The variant part must follow the other fields in the record declaration.

To declare a record type with a variant part, use the following syntax.

```
type recordTypeName = record
       fieldList1: type1;
        ...
       fieldListn: typen;
       case tag: ordinalType of
          constantList1: (variant1);
          ...
          constantListn: (variantn);
     end;
```

The first part of the declaration - up to the reserved word case - is the same as that of a standard record type. The remainder of the declaration - from case to the optional final semicolon - is called the variant part. In the variant part,

- *tag* is optional and can be any valid identifier. If you omit *tag*, omit the colon (**:**) after it as well.
- *ordinalType* denotes an ordinal type.
- Each *constantList* is a constant denoting a value of type *ordinalType*, or a comma-delimited list of such constants. No value can be represented more than once in the combined *constantLists*.

- Each *variant* is a semicolon-delimited list of declarations resembling the *fieldList: type* constructions in the main part of the record type. That is, a variant has the form

```
fieldList1: type1;
  ...
fieldListn: typen;
```

where each *fieldList* is a valid identifier or comma-delimited list of identifiers, each type denotes a type, and the final semicolon is optional. The types must not be long strings, dynamic arrays, variants (that is, Variant types), or interfaces, nor can they be structured types that contain long strings, dynamic arrays, variants, or interfaces; but they can be pointers to these types.

Records with variant parts are complicated syntactically but deceptively simple semantically. The variant part of a record contains several variants which share the same space in memory. You can read or write to any field of any variant at any time; but if you write to a field in one variant and then to a field in another variant, you may be overwriting your own data. The tag, if there is one, functions as an extra field (of type *ordinalType*) in the non-variant part of the record.

Variant parts have two purposes. First, suppose you want to create a record type that has fields for different kinds of data, but you know that you will never need to use all of the fields in a single record instance. For example,

```
type
   TEmployee = record
      FirstName, LastName: string[40];
      BirthDate: TDate;
      case Salaried: Boolean of
        True: (AnnualSalary: Currency);
        False: (HourlyWage: Currency);
   end;
```

The idea here is that every employee has either a salary or an hourly wage, but not both. So when you create an instance of `TEmployee`, there is no reason to allocate enough memory for both fields. In this case, the only difference between the variants is in the field names, but the fields could just as easily have been of different types. Consider some more complicated examples:

```
type
  TPerson = record
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Citizen: Boolean of
        True: (Birthplace: string[40]);
        False: (Country: string[20];
                EntryPort: string[20];
                EntryDate, ExitDate: TDate);
  end;

type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
         Rectangle: (Height, Width: Real);
         Triangle: (Side1, Side2, Angle: Real);
         Circle: (Radius: Real);
         Ellipse, Other: ();
  end;
```

For each record instance, the compiler allocates enough memory to hold all the fields in the largest variant. The optional tag and the *constantLists* (like `Rectangle`, `Triangle`, and so forth in the last example) play no role in the way the compiler manages the fields; they are there only for the convenience of the programmer.

The second reason for variant parts is that they let you treat the same data as belonging to different types, even in cases where the compiler would not allow a typecast. For example, if you have a 64-bit Real as the first field in one variant and a 32-bit Integer as the first field in another, you can assign a value to the Real field and then read back the first 32 bits of it as the value of the Integer field (passing it, say, to a function that requires integer parameters).

File Types

A file a sequence of elements of the same type. Standard I/O routines use the predefined `TextFile` or `Text` type, which represents a file containing characters organized into lines. For more information about file input and output, see Standard routines and I/O.

To declare a file type, use the syntax

type *fileTypeName* = file of *type*

where *fileTypeName* is any valid identifier and type is a fixed-size type. Pointer types - whether implicit or explicit - are not allowed, so a file cannot contain dynamic arrays, long strings, classes, objects, pointers, variants, other files, or structured types that contain any of these.

For example,

```
type
   PhoneEntry = record
     FirstName, LastName: string[20];
     PhoneNumber: string[15];
     Listed: Boolean;
   end;
   PhoneList = file of PhoneEntry;
```

declares a file type for recording names and telephone numbers.

You can also use the `file of ...` construction directly in a variable declaration. For example,

```
var List1: file of PhoneEntry;
```

The word file by itself indicates an untyped file:

```
var DataFile: file;
```

For more information, see Untyped files.

Files are not allowed in arrays or records.

Pointers and Pointer Types

A pointer is a variable that denotes a memory address. When a pointer holds the address of another variable, we say that it points to the location of that variable in memory or to the data stored there. In the case of an array or other structured type, a pointer holds the address of the first element in the structure. If that address is already taken, then the pointer holds the address to the first element.

Pointers are typed to indicate the kind of data stored at the addresses they hold. The general-purpose Pointer type can represent a pointer to any data, while more specialized pointer types reference only specific types of data. Pointers occupy four bytes of memory.

This topic contains information on the following:

- General overview of pointer types.
- Declaring and using the pointer types supported by Delphi.

Overview of pointers

To see how pointers work, look at the following example.

```
1         var
2           X, Y: Integer;  // X and Y are Integer variables
3           P: ^Integer     // P points to an Integer
4           begin
5             X := 17;      // assign a value to X
6             P := @X;      // assign the address of X to P
7             Y := P^;      // dereference P; assign the result to Y
8           end;
```

Line 2 declares `X` and `Y` as variables of type Integer. Line 3 declares `P` as a pointer to an Integer value; this means that `P` can point to the location of `X` or `Y`. Line 5 assigns a value to `X`, and line 6 assigns the address of `X` (denoted by `@X`) to `P`. Finally, line 7 retrieves the value at the location pointed to by `P` (denoted by `^P`) and assigns it to `Y`. After this code executes, `X` and `Y` have the same value, namely 17.

The **@** operator, which we have used here to take the address of a variable, also operates on functions and procedures. For more information, see The @ operator and Procedural types in statements and expressions.

The symbol **^** has two purposes, both of which are illustrated in our example. When it appears before a type identifier

^typeName

it denotes a type that represents pointers to variables of type *typeName*. When it appears after a pointer variable

*pointer^*

it dereferences the pointer; that is, it returns the value stored at the memory address held by the pointer.

Our example may seem like a roundabout way of copying the value of one variable to another - something that we could have accomplished with a simple assignment statement. But pointers are useful for several reasons. First, understanding pointers will help you to understand the Delphi language, since pointers often operate behind the scenes in code where they don't appear explicitly. Any data type that requires large, dynamically allocated blocks of memory uses pointers. Long-string variables, for instance, are implicitly pointers, as are class instance variables. Moreover, some advanced programming techniques require the use of pointers.

Finally, pointers are sometimes the only way to circumvent Delphi's strict data typing. By referencing a variable with an all-purpose Pointer, casting the Pointer to a more specific type, and then dereferencing it, you can treat the data

stored by any variable as if it belonged to any type. For example, the following code assigns data stored in a real variable to an integer variable.

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

Of course, reals and integers are stored in different formats. This assignment simply copies raw binary data from `R` to `I`, without converting it.

In addition to assigning the result of an **@** operation, you can use several standard routines to give a value to a pointer. The `New` and `GetMem` procedures assign a memory address to an existing pointer, while the `Addr` and `Ptr` functions return a pointer to a specified address or variable.

Dereferenced pointers can be qualified and can function as qualifiers, as in the expression `P1^.Data^`.

The reserved word nil is a special constant that can be assigned to any pointer. When nil is assigned to a pointer, the pointer doesn't reference anything.

Pointer Types

You can declare a pointer to any type, using the syntax

type *pointerTypeName* = ^*type*

When you define a record or other data type, it's a common practice also to define a pointer to that type. This makes it easy to manipulate instances of the type without copying large blocks of memory.

Standard pointer types exist for many purposes. The most versatile is Pointer, which can point to data of any kind. But a Pointer variable cannot be dereferenced; placing the **^** symbol after a Pointer variable causes a compilation error. To access the data referenced by a Pointer variable, first cast it to another pointer type and then dereference it.

Character Pointers

The fundamental types PAnsiChar and PWideChar represent pointers to AnsiChar and WideChar values, respectively. The generic PChar represents a pointer to a Char (that is, in its current implementation, to an AnsiChar). These character pointers are used to manipulate null-terminated strings. (See Working with null-terminated strings.)

Type-checked Pointers

The `$T` compiler directive controls the types of pointer values generated by the **@** operator. This directive takes the form of:

```
{$T+} or {$T-}
```

In the {$T-} state, the result type of the @ operator is always an untyped pointer that is compatible with all other pointer types. When @ is applied to a variable reference in the {$T+} state, the type of the result is ^T, where T is compatible only with pointers to the type of the variable.

Other Standard Pointer Types

The System and SysUtils units declare many standard pointer types that are commonly used.

Selected pointer types declared in System and SysUtils

Pointer type	Points to variables of type
PAnsiString, PString	AnsiString
PByteArray	TByteArray (declared in SysUtils). Used to typecast dynamically allocated memory for array access.
PCurrency, PDouble, PExtended, PSingle	Currency, Double, Extended, Single
PInteger	Integer
POleVariant	OleVariant
PShortString	ShortString. Useful when porting legacy code that uses the old PString type.
PTextBuf	TTextBuf (declared in SysUtils). TTextBuf is the internal buffer type in a TTextRec file record.)
PVarRec	TVarRec (declared in System)
PVariant	Variant
PWideString	WideString
PWordArray	TWordArray (declared in SysUtils). Used to typecast dynamically allocated memory for arrays of 2-byte values.

Procedural Types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions.

About Procedural Types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions. For example, suppose you define a function called `Calc` that takes two integer parameters and returns an integer:

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

You can assign the `Calc` function to the variable `F`:

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

If you take any procedure or function heading and remove the identifier after the word procedure or function, what's left is the name of a procedural type. You can use such type names directly in variable declarations (as in the previous example) or to declare new types:

```
type
  TIntegerFunction = function: Integer;
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFunc = function(X: Double): Double;
var
  F: TIntegerFunction;   { F is a parameterless function that returns an integer }
  Proc: TProcedure;      { Proc is a parameterless procedure }
  SP: TStrProc;          { SP is a procedure that takes a string parameter }
  M: TMathFunc;          { M is a function that takes a Double (real) parameter and returns
a Double }

  procedure FuncProc(P: TIntegerFunction);  { FuncProc is a procedure whose only parameter
is a parameterless integer-valued function }
```

On Win32, the variables shown in the previous example are all procedure pointers - that is, pointers to the address of a procedure or function. On the .NET platform, procedural types are implemented as delegates. If you want to reference a method of an instance object (see Classes and objects), you need to add the words `of object` to the procedural type name. For example

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

These types represent method pointers. A method pointer is really a pair of pointers; the first stores the address of a method, and the second stores a reference to the object the method belongs to. Given the declarations

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

```
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
      ...
  end;
var
  MainForm: TMainForm;
  OnClick: TNotifyEvent
```

we could make the following assignment.

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

Two procedural types are compatible if they have

- the same calling convention,
- the same return value (or no return value), and
- the same number of parameters, with identically typed parameters in corresponding positions. (Parameter names do not matter.)

Procedure pointer types are always incompatible with method pointer types. The value nil can be assigned to any procedural type.

Nested procedures and functions (routines declared within other routines) cannot be used as procedural values, nor can predefined procedures and functions. If you want to use a predefined routine like `Length` as a procedural value, write a wrapper for it:

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

Procedural Types in Statements and Expressions

When a procedural variable is on the left side of an assignment statement, the compiler expects a procedural value on the right. The assignment makes the variable on the left a pointer to the function or procedure indicated on the right. In other contexts, however, using a procedural variable results in a call to the referenced procedure or function. You can even use a procedural variable to pass parameters:

```
var
  F: function(X: Integer): Integer;
  I: Integer;
  function SomeFunction(X: Integer): Integer;
    ...
  F := SomeFunction;    // assign SomeFunction to F
  I := F(4);            // call function; assign result to I
```

In assignment statements, the type of the variable on the left determines the interpretation of procedure or method pointers on the right. For example,

```
var
  F, G: function: Integer;
  I: Integer;
```

```
function SomeFunction: Integer;
  ...
F := SomeFunction;      // assign SomeFunction to F
G := F;                 // copy F to G
I := G;                 // call function; assign result to I
```

The first statement assigns a procedural value to `F`. The second statement copies that value to another variable. The third statement makes a call to the referenced function and assigns the result to `I`. Because `I` is an integer variable, not a procedural one, the last assignment actually calls the function (which returns an integer).

In some situations it is less clear how a procedural variable should be interpreted. Consider the statement

```
if F = MyFunction then ...;
```

In this case, the occurrence of `F` results in a function call; the compiler calls the function pointed to by `F`, then calls the function `MyFunction`, then compares the results. The rule is that whenever a procedural variable occurs within an expression, it represents a call to the referenced procedure or function. In a case where `F` references a procedure (which doesn't return a value), or where `F` references a function that requires parameters, the previous statement causes a compilation error. To compare the procedural value of `F` with `MyFunction`, use

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

`@F` converts `F` into an untyped pointer variable that contains an address, and `@MyFunction` returns the address of `MyFunction`.

To get the memory address of a procedural variable (rather than the address stored in it), use **@@**. For example, `@@F` returns the address of `F`.

The **@** operator can also be used to assign an untyped pointer value to a procedural variable. For example,

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

calls the `GetProcAddress` function and points `StrComp` to the result.

Any procedural variable can hold the value nil, which means that it points to nothing. But attempting to call a nil-valued procedural variable is an error. To test whether a procedural variable is assigned, use the standard function `Assigned`:

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

Variant Types

This topic discusses the use of variant data types.

Variants Overview

Sometimes it is necessary to manipulate data whose type varies or cannot be determined at compile time. In these cases, one option is to use variables and parameters of type Variant, which represent values that can change type at runtime. Variants offer greater flexibility but consume more memory than regular variables, and operations on them are slower than on statically bound types. Moreover, illicit operations on variants often result in runtime errors, where similar mistakes with regular variables would have been caught at compile time. You can also create custom variant types.

By default, Variants can hold values of any type except records, sets, static arrays, files, classes, class references, and pointers. In other words, variants can hold anything but structured types and pointers. They can hold interfaces, whose methods and properties can be accessed through them. (See Object interfaces.) They can hold dynamic arrays, and they can hold a special kind of static array called a variant array. (See Variant arrays.) Variants can mix with other variants and with integer, real, string, and Boolean values in expressions and assignments; the compiler automatically performs type conversions.

Variants that contain strings cannot be indexed. That is, if `V` is a variant that holds a string value, the construction `V[1]` causes a runtime error.

You can define custom Variants that extend the Variant type to hold arbitrary values. For example, you can define a Variant string type that allows indexing or that holds a particular class reference, record type, or static array. Custom Variant types are defined by creating descendants to the `TCustomVariantType` class.

Note: This, and almost all variant functionality, is implemented in the `Variants` unit.

A variant occupies 16 bytes of memory and consists of a type code and a value, or pointer to a value, of the type specified by the code. All variants are initialized on creation to the special value Unassigned. The special value `Null` indicates unknown or missing data.

The standard function `VarType` returns a variant's type code. The `varTypeMask` constant is a bit mask used to extract the code from `VarType`'s return value, so that, for example,

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

returns True if `V` contains a Double or an array of Double. (The mask simply hides the first bit, which indicates whether the variant holds an array.) The TVarData record type defined in the `System` unit can be used to typecast variants and gain access to their internal representation.

Variant Type Conversions

All integer, real, string, character, and Boolean types are assignment-compatible with Variant. Expressions can be explicitly cast as variants, and the `VarAsType` and `VarCast` standard routines can be used to change the internal representation of a variant. The following code demonstrates the use of variants and some of the automatic conversions performed when variants are mixed with other types.

```
var
   V1, V2, V3, V4, V5: Variant;
   I: Integer;
   D: Double;
   S: string;
```

```
begin
   V1 := 1;          { integer value }
   V2 := 1234.5678; { real value }
   V3 := 'Hello world!'; { string value }
   V4 := '1000';         { string value }
   V5 := V1 + V2 + V4;   { real value 2235.5678}
   I  := V1;             { I = 1 (integer value) }
   D  := V2;              { D = 1234.5678 (real value) }
   S  := V3;              { S = 'Hello world!' (string value) }
   I  := V4;              { I = 1000 (integer value) }
   S  := V5;              { S = '2235.5678' (string value) }
end;
```

The compiler performs type conversions according to the following rules.

Variant type conversion rules

Target	**integer**	**real**	**string**	**Boolean**
Source				
integer	converts integer formats	converts to real	converts to string representation	returns False if 0, True otherwise
real	rounds to nearest integer	converts real formats	converts to string representation using regional settings	returns False if 0, True otherwise
string	converts to integer, truncating if necessary; raises exception if string is not numeric	converts to real using regional settings; raises exception if string is not numeric	converts string/character formats	returns False if string is 'false' (noncase-sensitive) or a numeric string that evaluates to 0, True if string is 'true' or a nonzero numeric string; raises exception otherwise
character	same as string (above)	same as string (above)	same as string (above)	same as string (above)
Boolean	False = 0, True = 1 (255 if Byte)	False = 0, True = 1	False = '0', True = '1'	False = False, True = True
Unassigned	returns 0	returns 0	returns empty string	returns False
Null	raises exception	raises exception	raises exception	raises exception

Out-of-range assignments often result in the target variable getting the highest value in its range. Invalid variant operations, assignments or casts raise an EVariantError exception or an exception class decending from EVariantError.

Special conversion rules apply to the TDateTime type declared in the `System` unit. When a TDateTime is converted to any other type, it treated as a normal Double. When an integer, real, or Boolean is converted to a TDateTime, it is first converted to a Double, then read as a date-time value. When a string is converted to a TDateTime, it is interpreted as a date-time value using the regional settings. When an `Unassigned` value is converted to TDateTime, it is treated like the real or integer value 0. Converting a `Null` value to TDateTime raises an exception.

On the Win32 platform, if a variant references a COM interface, any attempt to convert it reads the object's default property and converts that value to the requested type. If the object has no default property, an exception is raised.

Variants in Expressions

All operators except **^**, is, and in take variant operands. Except for comparisons, which always return a Boolean result, any operation on a variant value returns a variant result. If an expression combines variants with statically-typed values, the statically-typed values are automatically converted to variants.

This is not true for comparisons, where any operation on a Null variant produces a `Null` variant. For example:

```
V := Null + 3;
```

assigns a `Null` variant to `V`. By default, comparisons treat the `Null` variant as a unique value that is less than any other value. For example:

```
if Null > -3 then ... else ...;
```

In this example, the else part of the if statement will be executed. This behavior can be changed by setting the `NullEqualityRule` and `NullMagnitudeRule` global variables.

Variant Arrays

You cannot assign an ordinary static array to a variant. Instead, create a variant array by calling either of the standard functions `VarArrayCreate` or `VarArrayOf`. For example,

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

creates a variant array of integers (of length 10) and assigns it to the variant `V`. The array can be indexed using `V[0]`, `V[1]`, and so forth, but it is not possible to pass a variant array element as a var parameter. Variant arrays are always indexed with integers.

The second parameter in the call to `VarArrayCreate` is the type code for the array's base type. For a list of these codes, see VarType. Never pass the code `varString` to `VarArrayCreate`; to create a variant array of strings, use `varOleStr`.

Variants can hold variant arrays of different sizes, dimensions, and base types. The elements of a variant array can be of any type allowed in variants except ShortString and AnsiString, and if the base type of the array is Variant, its elements can even be heterogeneous. Use the `VarArrayRedim` function to resize a variant array. Other standard routines that operate on variant arrays include `VarArrayDimCount`, `VarArrayLowBound`, `VarArrayHighBound`, `VarArrayRef`, `VarArrayLock`, and `VarArrayUnlock`.

Note: Variant arrays of custom variants are not supported, as instances of custom variants can be added to a `VarVariant` variant array.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, the entire array is copied. Don't perform such operations unnecessarily, since they are memory-inefficient.

OleVariant

The OleVariant type exists on both the Windows and Linux platforms. The main difference between *Variant* and OleVariant is that *Variant* can contain data types that only the current application knows what to do with. OleVariant can only contain the data types defined as compatible with OLE Automation which means that the data types that can be passed between programs or across the network without worrying about whether the other end will know how to handle the data.

When you assign a *Variant* that contains custom data (such as a Delphi string, or a one of the new custom variant types) to an OleVariant, the runtime library tries to convert the *Variant* into one of the OleVariant standard data types (such as a Delphi string converts to an OLE BSTR string). For example, if a variant containing an AnsiString is assigned to an OleVariant, the AnsiString becomes a WideString. The same is true when passing a *Variant* to an OleVariant function parameter.

Type Compatibility and Identity

To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include:

- Type identity
- Type compatibility
- Assignment compatibility

Type Identity

When one type identifier is declared using another type identifier, without qualification, they denote the same type. Thus, given the declarations

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

`T1`, `T2`, `T3`, `T4`, and Integer all denote the same type. To create distinct types, repeat the word type in the declaration. For example,

```
type TMyInteger = type Integer;
```

creates a new type called `TMyInteger` which is not identical to Integer.

Language constructions that function as type names denote a different type each time they occur. Thus the declarations

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

create two distinct types, `TS1` and `TS2`. Similarly, the variable declarations

```
var
  S1: string[10];
  S2: string[10];
```

create two variables of distinct types. To create variables of the same type, use

```
var S1, S2: string[10];
```

or

```
type MyString = string[10];
var
```

```
    S1: MyString;
    S2: MyString;
```

Type Compatibility

Every type is compatible with itself. Two distinct types are compatible if they satisfy at least one of the following conditions.

- They are both real types.
- They are both integer types.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types.
- Both are packed-string types with the same number of characters.
- One is a string type and the other is a string, packed-string, or Char type.
- One type is Variant and the other is an integer, real, string, character, or Boolean type.
- Both are class, class-reference, or interface types, and one type is derived from the other.
- One type is PChar or PWideChar and the other is a zero-based character array of the form `array[0..n]` of PChar or PWideChar.
- One type is Pointer (an untyped pointer) and the other is any pointer type.
- Both types are (typed) pointers to the same type and the `{$T+}` compiler directive is in effect.
- Both are procedural types with the same result type, the same number of parameters, and type-identity between parameters in corresponding positions.

Assignment Compatibility

Assignment-compatibility is not a symmetric relation. An expression of type T2 can be assigned to a variable of type T1 if the value of the expression falls in the range of T1 and at least one of the following conditions is satisfied.

- T1 and T2 are of the same type, and it is not a file type or structured type that contains a file type at any level.
- T1 and T2 are compatible ordinal types.
- T1 and T2 are both real types.
- T1 is a real type and T2 is an integer type.
- T1 is PChar, PWideChar or any string type and the expression is a string constant.
- T1 and T2 are both string types.
- T1 is a string type and T2 is a Char or packed-string type.
- T1 is a long string and T2 is PChar or PWideChar.
- T1 and T2 are compatible packed-string types.
- T1 and T2 are compatible set types.
- T1 and T2 are compatible pointer types.
- T1 and T2 are both class, class-reference, or interface types and T2 is a derived from T1.
- T1 is an interface type and T2 is a class type that implements T1.
- T1 is PChar or PWideChar and T2 is a zero-based character array of the form `array[0..n] of Char` (when T1 is PChar) or of WideChar (when T1 is PWideChar).

- T1 and T2 are compatible procedural types. (A function or procedure identifier is treated, in certain assignment statements, as an expression of a procedural type.)
- T1 is Variant and T2 is an integer, real, string, character, Boolean, interface type or OleVariant type.
- T1 is an OleVariant and T2 is an integer, real, string, character, Boolean, interface, or Variant type.
- T1 is an integer, real, string, character, or Boolean type and T2 is Variant or OleVariant.
- T1 is the `IUnknown` or `IDispatch` interface type and T2 is Variant or OleVariant. (The variant's type code must be `varEmpty`, `varUnknown`, or `varDispatch` if T1 is `IUnknown`, and `varEmpty` or `varDispatch` if T1 is `IDispatch`.)

Declaring Types

This topic describes the syntax of Delphi type declarations.

Type Declaration Syntax

A type declaration specifies an identifier that denotes a type. The syntax for a type declaration is

type *newTypeName* = *type*

where *newTypeName* is a valid identifier. For example, given the type declaration

```
type TMyString = string;
```

you can make the variable declaration

```
var S: TMyString;
```

A type identifier's scope doesn't include the type declaration itself (except for pointer types). So you cannot, for example, define a record type that uses itself recursively.

When you declare a type that is identical to an existing type, the compiler treats the new type identifier as an alias for the old one. Thus, given the declarations

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

`X` and `Y` are of the same type; at runtime, there is no way to distinguish `TValue` from Real. This is usually of little consequence, but if your purpose in defining a new type is to utilize runtime type informationfor example, to associate a property editor with properties of a particular type - the distinction between 'different name' and 'different type' becomes important. In this case, use the syntax

`type` *newTypeName* = type*type*

For example,

```
type TValue = type Real;
```

forces the compiler to create a new, distinct type called `TValue`.

For `var` parameters, types of formal and actual must be identical. For example,

```
type
  TMyType = type Integer
  procedure p(var t:TMyType);
  begin
  end;

procedure x;
var
  m: TMyType;
  i: Integer;
begin
```

```
  p(m); // Works
  p(i); // Error! Types of formal and actual must be identical.
end;
```

Note: This only applies to `var` parameters, not to `const` or by-value parameters.

Variables

A variable is an identifier whose value can change at runtime. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold.

Declaring Variables

The basic syntax for a variable declaration is

var *identifierList* : *type;*

where *identifierList* is a comma-delimited list of valid identifiers and type is any valid type. For example,

```
var I: Integer;
```

declares a variable `I` of type Integer, while

```
var X, Y: Real;
```

declares two variables - `X` and `Y` - of type Real.

Consecutive variable declarations do not have to repeat the reserved word var:

```
var
   X, Y, Z: Double;
   I, J, K: Integer;
   Digit: 0..9;
   Okay: Boolean;
```

Variables declared within a procedure or function are sometimes called local, while other variables are called global. Global variables can be initialized at the same time they are declared, using the syntax

var *identifier: type* = *constantExpression;*

where *constantExpression* is any constant expression representing a value of type *type*. Thus the declaration

```
var I: Integer = 7;
```

is equivalent to the declaration and statement

```
var I: Integer;
    ...
I := 7;
```

Multiple variable declarations (such as `var X, Y, Z: Real;`) cannot include initializations, nor can declarations of variant and file-type variables.

If you don't explicitly initialize a global variable, the compiler initializes it to 0. Local variables, in contrast, cannot be initialized in their declarations and their contents are undefined until a value is assigned to them.

When you declare a variable, you are allocating memory which is freed automatically when the variable is no longer used. In particular, local variables exist only until the program exits from the function or procedure in which they are declared. For more information about variables and memory management, see Memory management.

Absolute Addresses

You can create a new variable that resides at the same address as another variable. To do so, put the directive absolute after the type name in the declaration of the new variable, followed by the name of an existing (previously declared) variable. For example,

```
var
   Str: string[32];
   StrLen: Byte absolute Str;
```

specifies that the variable `StrLen` should start at the same address as `Str`. Since the first byte of a short string contains the string's length, the value of `StrLen` is the length of `Str`.

You cannot initialize a variable in an absolute declaration or combine absolute with any other directives.

Dynamic Variables

You can create dynamic variables by calling the `GetMem` or `New` procedure. Such variables are allocated on the heap and are not managed automatically. Once you create one, it is your responsibility ultimately to free the variable's memory; use `FreeMem` to destroy variables created by `GetMem` and `Dispose` to destroy variables created by `New`. Other standard routines that operate on dynamic variables include `ReallocMem`, `AllocMem`, `Initialize`, `Finalize`, `StrAlloc`, and `StrDispose`.

Long strings, wide strings, dynamic arrays, variants, and interfaces are also heap-allocated dynamic variables, but their memory is managed automatically.

Thread-local Variables

Thread-local (or thread) variables are used in multithreaded applications. A thread-local variable is like a global variable, except that each thread of execution gets its own private copy of the variable, which cannot be accessed from other threads. Thread-local variables are declared with threadvar instead of var. For example,

```
threadvar X: Integer;
```

Thread-variable declarations

- cannot occur within a procedure or function.
- cannot include initializations.
- cannot specify the absolute directive.

Dynamic variables that are ordinarily managed by the compilerlong strings, wide strings, dynamic arrays, variants, and interfacescan be declared with threadvar, but the compiler does not automatically free the heap-allocated memory created by each thread of execution. If you use these data types in thread variables, it is your responsibility to dispose of their memory from within the thread, before the thread terminates. For example,

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  ...
S := '';  // free the memory used by S
```

Note: Use of such constructs is discouraged.

You can free a variant by setting it to `Unassigned` and an interface or dynamic array by setting it to nil.

Declared Constants

Several different language constructions are referred to as 'constants'. There are numeric constants (also called numerals) like 17, and string constants (also called character strings or string literals) like 'Hello world!'. Every enumerated type defines constants that represent the values of that type. There are predefined constants like True, False, and nil. Finally, there are constants that, like variables, are created individually by declaration.

Declared constants are either *true constants* or *typed constants*. These two kinds of constant are superficially similar, but they are governed by different rules and used for different purposes.

True Constants

A true constant is a declared identifier whose value cannot change. For example,

```
const MaxValue = 237;
```

declares a constant called `MaxValue` that returns the integer 237. The syntax for declaring a true constant is

const *identifier* = *constantExpression*

where identifier is any valid identifier and *constantExpression* is an expression that the compiler can evaluate without executing your program.

If *constantExpression* returns an ordinal value, you can specify the type of the declared constant using a value typecast. For example

```
const MyNumber = Int64(17);
```

declares a constant called `MyNumber`, of type Int64, that returns the integer 17. Otherwise, the type of the declared constant is the type of the *constantExpression*.

- If *constantExpression* is a character string, the declared constant is compatible with any string type. If the character string is of length 1, it is also compatible with any character type.
- If *constantExpression* is a real, its type is Extended. If it is an integer, its type is given by the table below.

Types for integer constants

Range of constant(hexadecimal)	Range of constant(decimal)	Type
-$8000000000000000..-$80000001	-2^63..-2147483649	Int64
-$80000000..-$8001	-2147483648..-32769	Integer
-$8000..-$81	-32768..-129	Smallint
-$80..-1	-128..-1	Shortint
0..$7F	0..127	0..127
$80..$FF	128..255	Byte
$0100..$7FFF	256..32767	0..32767
$8000..$FFFF	32768..65535	Word
$10000..$7FFFFFFF	65536..2147483647	0..2147483647
$80000000..$FFFFFFFF	2147483648..4294967295	Cardinal
$100000000..$7FFFFFFFFFFFFFFF	4294967296..2^631	Int64

Here are some examples of constant declarations:

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

Constant Expressions

A constant expression is an expression that the compiler can evaluate without executing the program in which it occurs. Constant expressions include numerals; character strings; true constants; values of enumerated types; the special constants True, False, and nil; and expressions built exclusively from these elements with operators, typecasts, and set constructors. Constant expressions cannot include variables, pointers, or function calls, except calls to the following predefined functions:

Abs	High	Low	Pred	Succ
Chr	Length	Odd	Round	Swap
Hi	Lo	Ord	SizeOf	Trunc

This definition of a constant expression is used in several places in Delphi's syntax specification. Constant expressions are required for initializing global variables, defining subrange types, assigning ordinalities to values in enumerated types, specifying default parameter values, writing case statements, and declaring both true and typed constants.

Examples of constant expressions:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

Resource Strings

Resource strings are stored as resources and linked into the executable or library so that they can be modified without recompiling the program.

Resource strings are declared like other true constants, except that the word const is replaced by resourcestring. The expression to the right of the **=** symbol must be a constant expression and must return a string value. For example,

```
resourcestring
  CreateError = 'Cannot create file %s';
  OpenError = 'Cannot open file %s';
  LineTooLong = 'Line too long';
  ProductName = 'Borland Rocks';
  SomeResourceString = SomeTrueConstant;
```

Typed Constants

Typed constants, unlike true constants, can hold values of array, record, procedural, and pointer types. Typed constants cannot occur in constant expressions.

Declare a typed constant like this:

const *identifier: type* = *value*

where identifier is any valid identifier, type is any type except files and variants, and value is an expression of type type. For example,

```
const Max: Integer = 100;
```

In most cases, value must be a constant expression; but if type is an array, record, procedural, or pointer type, special rules apply.

Array Constants

To declare an array constant, enclose the values of the array's elements, separated by commas, in parentheses at the end of the declaration. These values must be represented by constant expressions. For example,

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

declares a typed constant called `Digits` that holds an array of characters.

Zero-based character arrays often represent null-terminated strings, and for this reason string constants can be used to initialize character arrays. So the previous declaration can be more conveniently represented as

```
const Digits: array[0..9] of Char = '0123456789';
```

To define a multidimensional array constant, enclose the values of each dimension in a separate set of parentheses, separated by commas. For example,

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

creates an array called `Maze` where

```
Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
```

```
Maze[1,1,0] = 6
Maze[1,1,1] = 7
```

Array constants cannot contain file-type values at any level.

Record Constants

To declare a record constant, specify the value of each field - as `fieldName: value`, with the field assignments separated by semicolons - in parentheses at the end of the declaration. The values must be represented by constant expressions. The fields must be listed in the order in which they appear in the record type declaration, and the tag field, if there is one, must have a value specified; if the record has a variant part, only the variant selected by the tag field can be assigned values.

Examples:

```
type
  TPoint = record
     X, Y: Single;
  end;
  TVector = array[0..1] of TPoint;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

Record constants cannot contain file-type values at any level.

Procedural Constants

To declare a procedural constant, specify the name of a function or procedure that is compatible with the declared type of the constant. For example,

```
function Calc(X, Y: Integer): Integer;
begin
  ...
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

Given these declarations, you can use the procedural constant `MyFunction` in a function call:

```
I := MyFunction(5, 7)
```

You can also assign the value nil to a procedural constant.

Pointer Constants

When you declare a pointer constant, you must initialize it to a value that can be resolvedat least as a relative addressat compile time. There are three ways to do this: with the **@** operator, with nil, and (if the constant is of type

PChar or PWideChar) with a string literal. For example, if `I` is a global variable of type Integer, you can declare a constant like

```
const PI: ^Integer = @I;
```

The compiler can resolve this because global variables are part of the code segment. So are functions and global constants:

```
const PF: Pointer = @MyFunction;
```

Because string literals are allocated as global constants, you can initialize a PChar constant with a string literal:

```
const WarningStr: PChar = 'Warning!';
```

Procedures and Functions

This section describes the syntax of function and procedure declarations.

Procedures and Functions

This topic covers the following items:

- Declaring procedures and functions
- Calling conventions
- Forward and interface declarations
- Declaration of external routines
- Overloading procedures and functions
- Local declarations and nested routines

About Procedures and Functions

Procedures and functions, referred to collectively as *routines*, are self-contained statement blocks that can be called from different locations in a program. A function is a routine that returns a value when it executes. A procedure is a routine that does not return a value.

Function calls, because they return a value, can be used as expressions in assignments and operations. For example,

```
I := SomeFunction(X);
```

calls `SomeFunction` and assigns the result to `I`. Function calls cannot appear on the left side of an assignment statement.

Procedure calls - and, when extended syntax is enabled (`{$X+}`), function calls - can be used as complete statements. For example,

```
DoSomething;
```

calls the `DoSomething` routine; if `DoSomething` is a function, its return value is discarded.

Procedures and functions can call themselves recursively.

Declaring Procedures and Functions

When you declare a procedure or function, you specify its name, the number and type of parameters it takes, and, in the case of a function, the type of its return value; this part of the declaration is sometimes called the prototype, heading, or header. Then you write a block of code that executes whenever the procedure or function is called; this part is sometimes called the routine's body or block.

Procedure Declarations

A procedure declaration has the form

```
procedure procedureName(parameterList); directives;
  localDeclarations;
begin
  statements
end;
```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that execute when the procedure is called, and *(parameterList)*, *directives;*, and *localDeclarations;* are optional.

Here is an example of a procedure declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

Given this declaration, you can call the `NumString` procedure like this:

```
NumString(17, MyString);
```

This procedure call assigns the value '17' to `MyString` (which must be a string variable).

Within a procedure's statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the procedure. You can also use the parameter names from the parameter list (like `N` and `S` in the previous example); the parameter list defines a set of local variables, so don't try to redeclare the parameter names in the *localDeclarations* section. Finally, you can use any identifiers within whose scope the procedure declaration falls.

Function Declarations

A function declaration is like a procedure declaration except that it specifies a return type and a return value. Function declarations have the form

```
function functionName(parameterList): returnType; directives;
  localDeclarations;
begin
  statements
end;
```

where *functionName* is any valid identifier, *returnType* is a type identifier, statements is a sequence of statements that execute when the function is called, and *(parameterList)*, *directives;*, and *localDeclarations;* are optional.

The function's statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function's return value, as does the predefined variable `Result`.

As long as extended syntax is enabled (`{$X+}`), Result is implicitly declared in every function. Do not try to redeclare it.

For example,

```
function WF: Integer;
begin
```

```
  WF := 17;
end;
```

defines a constant function called `WF` that takes no parameters and always returns the integer value 17. This declaration is equivalent to

```
function WF: Integer;
begin
  Result := 17;
end;
```

Here is a more complicated function declaration:

```
function Max(A: array of Real; N: Integer): Real;
var
X: Real;
I: Integer;
begin
  X := A[0];
  for I := 1 to N - 1 do
    if X < A[I] then X := A[I];
  Max := X;
end;
```

You can assign a value to `Result` or to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to `Result` or to the function name becomes the function's return value. For example,

```
function Power(X: Real; Y: Integer): Real;
var
  I: Integer;
begin
  Result := 1.0;
  I := Y;
  while I > 0 do
   begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
   end;
end;
```

`Result` and the function name always represent the same value. Hence

```
function MyFunction: Integer;
begin
  MyFunction := 5;
  Result := Result * 2;
  MyFunction := Result + 1;
end;
```

returns the value 11. But `Result` is not completely interchangeable with the function name. When the function name appears on the left side of an assignment statement, the compiler assumes that it is being used (like `Result`) to

track the return value; when the function name appears anywhere else in the statement block, the compiler interprets it as a recursive call to the function itself. `Result`, on the other hand, can be used as a variable in operations, typecasts, set constructors, indexes, and calls to other routines.

If the function exits without assigning a value to `Result` or the function name, then the function's return value is undefined.

Calling Conventions

When you declare a procedure or function, you can specify a calling convention using one of the directives register, pascal, cdecl, stdcall, and safecall. For example,

```
function MyFunction(X, Y: Real): Real; cdecl;
```

Calling conventions determine the order in which parameters are passed to the routine. They also affect the removal of parameters from the stack, the use of registers for passing parameters, and error and exception handling. The default calling convention is register.

- The register and pascal conventions pass parameters from left to right; that is, the left most parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The cdecl, stdcall, and safecall conventions pass parameters from right to left.
- For all conventions except cdecl, the procedure or function removes parameters from the stack upon returning. With the cdecl convention, the caller removes parameters from the stack when the call returns.
- The register convention uses up to three CPU registers to pass parameters, while the other conventions pass all parameters on the stack.
- The safecall convention implements exception 'firewalls.' On Win32, this implements interprocess COM error notification.

The table below summarizes calling conventions.

Calling conventions

Directive	Parameter order	Clean-up	Passes parameters in registers?
register	Left-to-right	Routine	Yes
pascal	Left-to-right	Routine	No
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Routine	No
safecall	Right-to-left	Routine	No

The default register convention is the most efficient, since it usually avoids creation of a stack frame. (Access methods for published properties must use register.) The cdecl convention is useful when you call functions from shared libraries written in C or C++, while stdcall and safecall are recommended, in general, for calls to external code. On Win32, the operating system APIs are stdcall and safecall. Other operating systems generally use cdecl. (Note that stdcall is more efficient than cdecl.)

The safecall convention must be used for declaring dual-interface methods. The pascal convention is maintained for backward compatibility.

The directives near, far, and export refer to calling conventions in 16-bit Windows programming. They have no effect in Win32, or in .NET applications and are maintained for backward compatibility only.

Forward and Interface Declarations

The forward directive replaces the block, including local variable declarations and statements, in a procedure or function declaration. For example,

```
function Calculate(X, Y: Integer): Real; forward;
```

declares a function called `Calculate`. Somewhere after the forward declaration, the routine must be redeclared in a defining declaration that includes a block. The defining declaration for `Calculate` might look like this:

```
function Calculate;
  ... { declarations }
begin
  ... { statement block }
end;
```

Ordinarily, a defining declaration does not have to repeat the routine's parameter list or return type, but if it does repeat them, they must match those in the forward declaration exactly (except that default parameters can be omitted). If the forward declaration specifies an overloaded procedure or function, then the defining declaration must repeat the parameter list.

A forward declaration and its defining declaration must appear in the same type declaration section. That is, you can't add a new section (such as a var section or const section) between the forward declaration and the defining declaration. The defining declaration can be an external or assembler declaration, but it cannot be another forward declaration.

The purpose of a forward declaration is to extend the scope of a procedure or function identifier to an earlier point in the source code. This allows other procedures and functions to call the forward-declared routine before it is actually defined. Besides letting you organize your code more flexibly, forward declarations are sometimes necessary for mutual recursions.

The forward directive has no effect in the interface section of a unit. Procedure and function headers in the interface section behave like forward declarations and must have defining declarations in the implementation section. A routine declared in the interface section is available from anywhere else in the unit and from any other unit or program that uses the unit where it is declared.

External Declarations

The external directive, which replaces the block in a procedure or function declaration, allows you to call routines that are compiled separately from your program. External routines can come from object files or dynamically loadable libraries.

When importing a C function that takes a variable number of parameters, use the varargs directive. For example,

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

The varargs directive works only with external routines and only with the cdecl calling convention.

Linking to Object Files

To call routines from a separately compiled object file, first link the object file to your application using the `$L` (or `$LINK`) compiler directive. For example,

```
{$L BLOCK.OBJ}
```

links BLOCK.OBJ into the program or unit in which it occurs. Next, declare the functions and procedures that you want to call:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Now you can call the `MoveWord` and `FillWord` routines from BLOCK.OBJ.

On the Win32 platform, declarations like the ones above are frequently used to access external routines written in assembly language. You can also place assembly-language routines directly in your Delphi source code.

Importing Functions from Libraries

To import routines from a dynamically loadable library (.DLL), attach a directive of the form

`external` *stringConstant*;

to the end of a normal procedure or function header, where *stringConstant* is the name of the library file in single quotation marks. For example, on Win32

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

imports a function called `SomeFunction` from strlib.dll.

You can import a routine under a different name from the one it has in the library. If you do this, specify the original name in the external directive:

`external` *stringConstant1*name*stringConstant2;*

where the first *stringConstant* gives the name of the library file and the second *stringConstant* is the routine's original name.

The following declaration imports a function from user32.dll (part of the Win32 API).

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer; stdcall;
external 'user32.dll' name 'MessageBoxA';
```

The function's original name is `MessageBoxA`, but it is imported as `MessageBox`.

Instead of a name, you can use a number to identify the routine you want to import:

external*stringConstant*index*integerConstant;*

where *integerConstant* is the routine's index in the export table.

In your importing declaration, be sure to match the exact spelling and case of the routine's name. Later, when you call the imported routine, the name is case-insensitive.

Overloading Procedures and Functions

You can declare more than one routine in the same scope with the same name. This is called overloading. Overloaded routines must be declared with the overload directive and must have distinguishing parameter lists. For example, consider the declarations

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X/Y;
```

```
end

function Divide(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

These declarations create two functions, both called `Divide`, that take parameters of different types. When you call `Divide`, the compiler determines which function to invoke by looking at the actual parameters passed in the call. For example, `Divide(6.0, 3.0)` calls the first `Divide` function, because its arguments are real-valued.

You can pass to an overloaded routine parameters that are not identical in type with those in any of the routine's declarations, but that are assignment-compatible with the parameters in more than one declaration. This happens most frequently when a routine is overloaded with different integer types or different real types - for example,

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

In these cases, when it is possible to do so without ambiguity, the compiler invokes the routine whose parameters are of the type with the smallest range that accommodates the actual parameters in the call. (Remember that real-valued constant expressions are always of type Extended.)

Overloaded routines must be distinguished by the number of parameters they take or the types of their parameters. Hence the following pair of declarations causes a compilation error.

```
function Cap(S: string): string; overload;
  ...
procedure Cap(var Str: string); overload;
  ...
```

But the declarations

```
function Func(X: Real; Y: Integer): Real; overload;
  ...
function Func(X: Integer; Y: Real): Real; overload;
  ...
```

are legal.

When an overloaded routine is declared in a forward or interface declaration, the defining declaration must repeat the routine's parameter list.

The compiler can distinguish between overloaded functions that contain AnsiString/PChar and WideString/WideChar parameters in the same parameter position. String constants or literals passed into such an overload situation are translated into the native string or character type, which is AnsiString/PChar.

```
procedure test(const S: String);  overload;
procedure test(const W: WideString); overload;
var
  a: string;
  b: widestring;
begin
  a := 'a';
  b := 'b';
```

```
  test(a);    // calls String version
  test(b);    // calls WideString version
  test('abc');    // calls String version
  test(WideString('abc'));   // calls widestring version
end;
```

Variants can also be used as parameters in overloaded function declarations. Variant is considered more general than any simple type. Preference is always given to exact type matches over variant matches. If a variant is passed into such an overload situation, and an overload that takes a variant exists in that parameter position, it is considered to be an exact match for the Variant type.

This can cause some minor side effects with float types. Float types are matched by size. If there is no exact match for the float variable passed to the overload call but a variant parameter is available, the variant is taken over any smaller float type.

For example:

```
procedure foo(i: integer); overload;
procedure foo(d: double); overload;
procedure foo(v: variant); overload;
var
  v: variant;
begin
  foo(1);       // integer version
  foo(v);       // variant version
  foo(1.2);     // variant version (float literals -&gt; extended precision)
end;
```

This example calls the variant version of `foo`, not the double version, because the 1.2 constant is implicitly an extended type and extended is not an exact match for double. Extended is also not an exact match for Variant, but Variant is considered a more general type (whereas double is a smaller type than extended).

```
foo(Double(1.2));
```

This typecast does not work. You should use typed consts instead.

```
const  d: double = 1.2;
begin
   foo(d);
end;
```

The above code works correctly, and calls the double version.

```
const  s: single = 1.2;
begin
  foo(s);
end;
```

The above code also calls the double version of `foo`. Single is a better fit to double than to variant.

When declaring a set of overloaded routines, the best way to avoid float promotion to variant is to declare a version of your overloaded function for each float type (Single, Double, Extended) along with the variant version.

If you use default parameters in overloaded routines, be careful not to introduce ambiguous parameter signatures.

You can limit the potential effects of overloading by qualifying a routine's name when you call it. For example, `Unit1.MyProcedure(X, Y)` can call only routines declared in `Unit1;` if no routine in `Unit1` matches the name and parameter list in the call, an error results.

Local Declarations

The body of a function or procedure often begins with declarations of local variables used in the routine's statement block. These declarations can also include constants, types, and other routines. The scope of a local identifier is limited to the routine where it is declared.

Nested Routines

Functions and procedures sometimes contain other functions and procedures within the local-declarations section of their blocks. For example, the following declaration of a procedure called `DoSomething` contains a nested procedure.

```
procedure DoSomething(S: string);
var
  X, Y: Integer;

procedure NestedProc(S: string);
begin
  ...
end;

begin
  ...
  NestedProc(S);
  ...
end;
```

The scope of a nested routine is limited to the procedure or function in which it is declared. In our example, `NestedProc` can be called only within `DoSomething`.

For real examples of nested routines, look at the DateTimeToString procedure, the ScanDate function, and other routines in the `SysUtils` unit.

Parameters

This topic covers the following items:

- Parameter semantics
- String parameters
- Array parameters
- Default parameters

About Parameters

Most procedure and function headers include a parameter list. For example, in the header

```
function Power(X: Real; Y: Integer): Real;
```

the parameter list is `(X: Real; Y: Integer)`.

A parameter list is a sequence of parameter declarations separated by semicolons and enclosed in parentheses. Each declaration is a comma-delimited series of parameter names, followed in most cases by a colon and a type identifier, and in some cases by the **=** symbol and a default value. Parameter names must be valid identifiers. Any declaration can be preceded by var, const, or out. Examples:

```
(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

The parameter list specifies the number, order, and type of parameters that must be passed to the routine when it is called. If a routine does not take any parameters, omit the identifier list and the parentheses in its declaration:

```
procedure UpdateRecords;
begin
  ...
end;
```

Within the procedure or function body, the parameter names (`X` and `Y` in the first example) can be used as local variables. Do not redeclare the parameter names in the local declarations section of the procedure or function body.

Parameter Semantics

Parameters are categorized in several ways:

- Every parameter is classified as value, variable, constant, or out. Value parameters are the default; the reserved words var, const, and out indicate variable, constant, and out parameters, respectively.
- Value parameters are always typed, while constant, variable, and out parameters can be either typed or untyped.
- Special rules apply to array parameters.

Files and instances of structured types that contain files can be passed only as variable (var) parameters.

Value and Variable Parameters

Most parameters are either value parameters (the default) or variable (var) parameters. Value parameters are passed by value, while variable parameters are passed by reference. To see what this means, consider the following functions.

```
function DoubleByValue(X: Integer): Integer;   // X is a value parameter
begin
  X := X * 2;
  Result := X;
end;

function DoubleByRef(var X: Integer): Integer;  // X is a variable parameter
begin
  X := X * 2;
  Result := X;
end;
```

These functions return the same result, but only the second one - `DoubleByRefcan` change the value of a variable passed to it. Suppose we call the functions like this:

```
var
  I, J, V, W: Integer;
begin
  I := 4;
  V := 4;
  J := DoubleByValue(I);    // J = 8, I = 4
  W := DoubleByRef(V);      // W = 8, V = 8
end;
```

After this code executes, the variable `I`, which was passed to `DoubleByValue`, has the same value we initially assigned to it. But the variable `V`, which was passed to `DoubleByRef`, has a different value.

A value parameter acts like a local variable that gets initialized to the value passed in the procedure or function call. If you pass a variable as a value parameter, the procedure or function creates a copy of it; changes made to the copy have no effect on the original variable and are lost when program execution returns to the caller.

A variable parameter, on the other hand, acts like a pointer rather than a copy. Changes made to the parameter within the body of a function or procedure persist after program execution returns to the caller and the parameter name itself has gone out of scope.

Even if the same variable is passed in two or more var parameters, no copies are made. This is illustrated in the following example.

```
procedure AddOne(var X, Y: Integer);
begin
  X := X + 1;
  Y := Y + 1;
end;

var I: Integer;
begin
  I := 1;
  AddOne(I, I);
end;
```

After this code executes, the value of `I` is 3.

If a routine's declaration specifies a var parameter, you must pass an assignable expression - that is, a variable, typed constant (in the `{$J+}` state), dereferenced pointer, field, or indexed variableto the routine when you call it. To use our previous examples, `DoubleByRef(7)` produces an error, although `DoubleByValue(7)` is legal.

Indexes and pointer dereferences passed in var parameters - for example, `DoubleByRef(MyArray[I])` - are evaluated once, before execution of the routine.

Constant Parameters

A constant (const) parameter is like a local constant or read-only variable. Constant parameters are similar to value parameters, except that you can't assign a value to a constant parameter within the body of a procedure or function, nor can you pass one as a var parameter to another routine. (But when you pass an object reference as a constant parameter, you can still modify the object's properties.)

Using const allows the compiler to optimize code for structured - and string-type parameters. It also provides a safeguard against unintentionally passing a parameter by reference to another routine.

Here, for example, is the header for the `CompareStr` function in the `SysUtils` unit:

```
function CompareStr(const S1, S2: string): Integer;
```

Because `S1` and `S2` are not modified in the body of CompareStr, they can be declared as constant parameters.

Out Parameters

An out parameter, like a variable parameter, is passed by reference. With an out parameter, however, the initial value of the referenced variable is discarded by the routine it is passed to. The out parameter is for output only; that is, it tells the function or procedure where to store output, but doesn't provide any input.

For example, consider the procedure heading

```
procedure GetInfo(out Info: SomeRecordType);
```

When you call `GetInfo`, you must pass it a variable of type `SomeRecordType`:

```
var MyRecord: SomeRecordType;
   ...
GetInfo(MyRecord);
```

But you're not using `MyRecord` to pass any data to the `GetInfo` procedure; `MyRecord` is just a container where you want `GetInfo` to store the information it generates. The call to `GetInfo` immediately frees the memory used by `MyRecord`, before program control passes to the procedure.

Out parameters are frequently used with distributed-object models like COM and CORBA. In addition, you should use out parameters when you pass an uninitialized variable to a function or procedure.

Untyped Parameters

You can omit type specifications when declaring var, const, and out parameters. (Value parameters must be typed.) For example,

```
procedure TakeAnything(const C);
```

declares a procedure called `TakeAnything` that accepts a parameter of any type. When you call such a routine, you cannot pass it a numeral or untyped numeric constant.

Within a procedure or function body, untyped parameters are incompatible with every type. To operate on an untyped parameter, you must cast it. In general, the compiler cannot verify that operations on untyped parameters are valid.

The following example uses untyped parameters in a function called `Equal` that compares a specified number of bytes of any two variables.

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N : Integer;
begin
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
    Equal := N = Size;
end;
```

Given the declarations

```
type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;
```

you could make the following calls to `Equal`:

```
Equal(Vec1, Vec2, SizeOf(TVector));      // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N);  // compare first N elements of Vec1 and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5);   // compare first 5 to last 5 elements of Vec1
Equal(Vec1[1], P, 4);                    // compare Vec1[1] to P.X and Vec1[2] to P.Y
```

String Parameters

When you declare routines that take short-string parameters, you cannot include length specifiers in the parameter declarations. That is, the declaration

```
procedure Check(S: string[20]);   // syntax error
```

causes a compilation error. But

```
type TString20 = string[20];
procedure Check(S: TString20);
```

is valid. The special identifier OpenString can be used to declare routines that take short-string parameters of varying length:

```
procedure Check(S: OpenString);
```

When the `{$H}` and `{$P+}` compiler directives are both in effect, the reserved word string is equivalent to OpenString in parameter declarations.

Short strings, OpenString, `$H`, and `$P` are supported for backward compatibility only. In new code, you can avoid these considerations by using long strings.

Array Parameters

When you declare routines that take array parameters, you cannot include index type specifiers in the parameter declarations. That is, the declaration

```
procedure Sort(A: array[1..10] of Integer)  // syntax error<
```

causes a compilation error. But

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

is valid. Another approach is to use open array parameters.

Since the Delphi language does not implement value semantics for dynamic arrays, 'value' parameters in routines do not represent a full copy of the dynamic array. In this example

```
type
  TDynamicArray = array of Integer;
  procedure p(Value: TDynamicArray);
  begin
    Value[0] := 1;
  end;

  procedure Run;
var
    a: TDynamicArray;
begin
  SetLength(a, 1);
  a[0] := 0;
  p(a);
  Writeln(a[0]); // Prints '1'
end;
```

Note that the assignment to `Value[0]` in routine `p` will modify the content of dynamic array of the caller, despite `Value` being a by-value parameter. If a full copy of the dynamic array is required, use the Copy standard procedure to create a value copy of the dynamic array.

Open Array Parameters

Open array parameters allow arrays of different sizes to be passed to the same procedure or function. To define a routine with an open array parameter, use the syntax `array of` *type* (rather than `array[X..Y] of` *type*) in the parameter declaration. For example,

```
function Find(A: array of Char): Integer;
```

declares a function called `Find` that takes a character array of any size and returns an integer.

Note: The syntax of open array parameters resembles that of dynamic array types, but they do not mean the same thing. The previous example creates a function that takes any array of Char elements, including (but not limited to) dynamic arrays. To declare parameters that must be dynamic arrays, you need to specify a type identifier:

```
type TDynamicCharArray = array of Char;
function Find(A: TDynamicCharArray): Integer;
```

Within the body of a routine, open array parameters are governed by the following rules.

- They are always zero-based. The first element is 0, the second element is 1, and so forth. The standard Low and High functions return 0 and Length1, respectively. The SizeOf function returns the size of the actual array passed to the routine.
- They can be accessed by element only. Assignments to an entire open array parameter are not allowed.
- They can be passed to other procedures and functions only as open array parameters or untyped var parameters. They cannot be passed to SetLength.
- Instead of an array, you can pass a variable of the open array parameter's base type. It will be treated as an array of length 1.

When you pass an array as an open array value parameter, the compiler creates a local copy of the array within the routine's stack frame. Be careful not to overflow the stack by passing large arrays.

The following examples use open array parameters to define a `Clear` procedure that assigns zero to each element in an array of reals and a `Sum` function that computes the sum of the elements in an array of reals.

```
procedure Clear(var A: array of Real);
var
   I: Integer;
begin
   for I := 0 to High(A) do A[I] := 0;
end;

function Sum(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Sum := S;
end;
```

When you call routines that use open array parameters, you can pass open array constructors to them.

Variant Open Array Parameters

Variant open array parameters allow you to pass an array of differently typed expressions to a single procedure or function. To define a routine with a variant open array parameter, specify `array of const` as the parameter's type. Thus

```
procedure DoSomething(A: array of const);
```

declares a procedure called `DoSomething` that can operate on heterogeneous arrays.

The `array of const` construction is equivalent to `array of TVarRec`. `TVarRec`, declared in the `System` unit, represents a record with a variant part that can hold values of integer, Boolean, character, real, string, pointer, class, class reference, interface, and variant types. `TVarRec`'s `VType` field indicates the type of each element in the array. Some types are passed as pointers rather than values; in particular, long strings are passed as Pointer and must be typecast to string.

The following example uses a variant open array parameter in a function that creates a string representation of each element passed to it and concatenates the results into a single string. The string-handling routines called in this function are defined in `SysUtils`.

```
function MakeStr(const Args: array of const): string
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
     with Args[I] do
        case VType of
            vtInteger:  Result := Result + IntToStr(VInteger);
            vtBoolean:  Result := Result + BoolToStr(VBoolean);
            vtChar:     Result := Result + VChar;
            vtExtended: Result := Result + FloatToStr(VExtended^);
            vtString:   Result := Result + VString^;
            vtPChar:    Result := Result + VPChar;
            vtObject:   Result := Result + VObject.ClassName;
            vtClass:    Result := Result + VClass.ClassName;
            vtAnsiString:  Result := Result + string(VAnsiString);
            vtCurrency:    Result := Result + CurrToStr(VCurrency^);
            vtVariant:     Result := Result + string(VVariant^);
            vtInt64:       Result := Result + IntToStr(VInt64^);
  end;
end;
```

We can call this function using an open array constructor. For example,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

returns the string `'test100 T3.14159TForm'`.

Default Parameters

You can specify default parameter values in a procedure or function heading. Default values are allowed only for typed const and value parameters. To provide a default value, end the parameter declaration with the **=** symbol followed by a constant expression that is assignment-compatible with the parameter's type.

For example, given the declaration

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

the following procedure calls are equivalent.

```
FillArray(MyArray);
          FillArray(MyArray, 0);
```

A multiple-parameter declaration cannot specify a default value. Thus, while

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

is legal,

```
function MyFunction(X, Y: Real = 3.5): Real;  // syntax error
```

is not.

Parameters with default values must occur at the end of the parameter list. That is, all parameters following the first declared default value must also have default values. So the following declaration is illegal.

```
procedure MyProcedure(I: Integer = 1; S: string);  // syntax error
```

Default values specified in a procedural type override those specified in an actual routine. Thus, given the declarations

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
  F: TResizer;
  N: Real;
```

the statements

```
F := Resizer;
F(N);
```

result in the values `(N, 1.0)` being passed to `Resizer`.

Default parameters are limited to values that can be specified by a constant expression. Hence parameters of a dynamic-array, procedural, class, class-reference, or interface type can have no value other than nil as their default. Parameters of a record, variant, file, static-array, or object type cannot have default values at all.

Default Parameters and Overloaded Functions

If you use default parameter values in an overloaded routine, avoid ambiguous parameter signatures. Consider, for example, the following.

```
procedure Confused(I: Integer); overload;
   ...
procedure Confused(I: Integer; J: Integer = 0); overload;
   ...
Confused(X);   //  Which procedure is called?
```

In fact, neither procedure is called. This code generates a compilation error.

Default Parameters in Forward and Interface Declarations

If a routine has a forward declaration or appears in the interface section of a unit, default parameter valuesif there are anymust be specified in the forward or interface declaration. In this case, the default values can be omitted from the defining (implementation) declaration; but if the defining declaration includes default values, they must match those in the forward or interface declaration exactly.

Calling Procedures and Functions

This topic covers the following items:

- Program control and routine parameters
- Open array constructors
- The inline directive

Program Control and Parameters

When you call a procedure or function, program control passes from the point where the call is made to the body of the routine. You can make the call using the routine's declared name (with or without qualifiers) or using a procedural variable that points to the routine. In either case, if the routine is declared with parameters, your call to it must pass parameters that correspond in order and type to the routine's parameter list. The parameters you pass to a routine are called actual parameters, while the parameters in the routine's declaration are called formal parameters.

When calling a routine, remember that

- expressions used to pass typed const and value parameters must be assignment-compatible with the corresponding formal parameters.
- expressions used to pass var and out parameters must be identically typed with the corresponding formal parameters, unless the formal parameters are untyped.
- only assignable expressions can be used to pass var and out parameters.
- if a routine's formal parameters are untyped, numerals and true constants with numeric values cannot be used as actual parameters.

When you call a routine that uses default parameter values, all actual parameters following the first accepted default must also use the default values; calls of the form `SomeFunction(,,X)` are not legal.

You can omit parentheses when passing all and only the default parameters to a routine. For example, given the procedure

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

the following calls are equivalent.

```
DoSomething();
          DoSomething;
```

Open Array Constructors

Open array constructors allow you to construct arrays directly within function and procedure calls. They can be passed only as open array parameters or variant open array parameters.

An open array constructor, like a set constructor, is a sequence of expressions separated by commas and enclosed in brackets.

For example, given the declarations

```
var I, J: Integer;
          procedure Add(A: array of Integer);
```

you could call the `Add` procedure with the statement

```
Add([5, 7, I, I + J]);
```

This is equivalent to

```
var Temp: array[0..3] of Integer;
            ...
          Temp[0] := 5;
          Temp[1] := 7;
          Temp[2] := I;
          Temp[3] := I + J;
          Add(Temp);
```

Open array constructors can be passed only as value or const parameters. The expressions in a constructor must be assignment-compatible with the base type of the array parameter. In the case of a variant open array parameter, the expressions can be of different types.

Using the inline Directive

The Delphi compiler allows functions and procedures to be tagged with the inline directive to improve performance. If the function or procedure meets certain criteria, the compiler will insert code directly, rather than generating a call. Inlining is a performance optimization that can result in faster code, but at the expense of space. Inlining always causes the compiler to produce a larger binary file. The inline directive is used in function and procedure declarations and definitions, like other directives.

```
procedure MyProc(x:Integer); inline;
begin
    // ...
end;

function MyFunc(y:Char) : String; inline;
begin
   // ..
end;
```

The inline directive is a suggestion to the compiler. There is no guarantee the compiler will inline a particular routine, as there are a number of circumstances where inlining cannot be done. The following list shows the conditions under which inlining does or does not occur:

- Inlining will not occur on any form of late-bound method. This includes virtual, dynamic, and message methods.
- Routines containing assembly code will not be inlined.
- Constructors and destructors will not be inlined.
- The main program block, unit initialization, and unit finalization blocks cannot be inlined.
- Routines that take open array parameters cannot be inlined.
- Code can be inlined within packages, however, inlining never occurs across package boundaries.
- No inlining will be done between units that are circularly dependent. This included indirect circular dependencies, for example, unit A uses unit B, and unit B uses unit C which in turn uses unit A. In this example, when compiling unit A, no code from unit B or unit C will be inlined in unit A.
- The compiler can inline code when a unit is in a circular dependency, as long as the code to be inlined comes from a unit outside the circular relationship. In the above example, if unit A also used unit D, code from unit D could be inlined in A, since it is not involved in the circular dependency.

- If a routine is defined in the interface section and it accesses symbols defined in the implementation section, that routine cannot be inlined.
- Routines in classes cannot be inlined if they access members with less (i.e. more restricted) visibility than the method itself. For example, if a public method accesses private symbols, it cannot be inlined.
- If a routine marked with inline uses external symbols from other units, all of those units must be listed in the uses statement, otherwise the routine cannot be inlined.
- Procedures and functions used in conditional expressions in while-do and repeat-until statements cannot be expanded inline.

If you modify the implementation of an inlined routine, you will cause all units that use that function to be recompiled. This is different from traditional rebuild rules, where rebuilds were triggered only by changes in the interface section of a unit.

The `{$INLINE}` compiler directive gives you finer control over inlining. The `{$INLINE}` directive can be used at the site of the inlined routine's definition, as well as at the call site. Below are the possible values and their meaning:

Value	Meaning at definition	Meaning at call site
`{$INLINE ON}` (default)	The routine is compiled as inlineable if it is tagged with the inline directive.	The routine will be expanded inline if possible.
`{$INLINE AUTO}`	Behaves like `{$INLINE ON}`, with the addition that routines not marked with inline will be inlined if their code size is less than or equal to 32 bytes.	`{$INLINE AUTO}` has no effect at the call site.
`{$INLINE OFF}`	The routine will not be marked as inlineable, even if it is tagged with inline.	The routine will not be expanded inline.

Classes and Objects

This section describes the object-oriented features of the Delphi language, such as the declaration and usage of class types.

Classes and Objects

This topic covers the following material:

- Declaration syntax of classes
- Inheritance and scope
- Visibility of class members
- Forward declarations and mutually dependent classes

Class Types

A class, or class type, defines a structure consisting of fields, methods, and properties. Instances of a class type are called objects. The fields, methods, and properties of a class are called its components or members.

- A field is essentially a variable that is part of an object. Like the fields of a record, a class' fields represent data items that exist in each instance of the class.
- A method is a procedure or function associated with a class. Most methods operate on objectsthat is, instances of a class. Some methods (called class methods) operate on class types themselves.
- A property is an interface to data associated with an object (often stored in a field). Properties have access specifiers, which determine how their data is read and modified. From other parts of a programoutside of the object itselfa property appears in most respects like a field.

Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Objects are created and destroyed by special methods called constructors and destructors.

A variable of a class type is actually a pointer that references an object. Hence more than one variable can refer to the same object. Like other pointers, class-type variables can hold the value nil. But you don't have to explicitly dereference a class-type variable to access the object it points to. For example, `SomeObject.Size := 100` assigns the value 100 to the `Size` property of the object referenced by `SomeObject`; you would not write this as `SomeObject^.Size := 100`.

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a program or unit, not in a procedure or function declaration.

A class type declaration has the form

```
type
   className = class (ancestorClass)
      memberList
   end;
```

where *className* is any valid identifier, *(ancestorClass)* is optional, and *memberList* declares members - that is, fields, methods, and properties - of the class. If you omit *(ancestorClass)*, then the new class inherits directly from the predefined TObject class. If you include *(ancestorClass)* and *memberList* is empty, you can omit end. A class type declaration can also include a list of interfaces implemented by the class; see Implementing Interfaces.

Delphi for .NET supports the additional features of sealed classes and abstract classes. A sealed class is one that cannot be extended through inheritance. This includes all .NET languages that might use the sealed class. Delphi for .NET also allows an entire class to be declared as abstract, even though it does not contain any abstract virtual methods. The class declaration syntax for Delphi for .NET is:

```
type
   className = class [abstract | sealed] (ancestorType)
       memberList
   end;
```

Methods appear in a class declaration as function or procedure headings, with no body. Defining declarations for each method occur elsewhere in the program.

For example, here is the declaration of the TMemoryStream class from the `Classes` unit.

```
    type TMemoryStream = class(TCustomMemoryStream)
           private
             FCapacity: Longint;
             procedure SetCapacity(NewCapacity: Longint);
           protected
             function Realloc(var NewCapacity: Longint): Pointer; virtual;
             property Capacity: Longint read FCapacity write SetCapacity;
           public
             destructor Destroy; override;
             procedure Clear;
             procedure LoadFromStream(Stream: TStream);
             procedure LoadFromFile(const FileName: string);
             procedure SetSize(NewSize: Longint); override;
             function Write(const Buffer; Count: Longint): Longint; override;
         end;
```

TMemoryStream descends from TCustomMemoryStream (in the `Classes` unit), inheriting most of its members. But it defines - or redefines - several methods and properties, including its destructor method, `Destroy`. Its constructor, `Create`, is inherited without change from TObject, and so is not redeclared. Each member is declared as private, protected, or public (this class has no published members). These terms are explained below.

Given this declaration, you can create an instance of TMemoryStream as follows:

```
var stream: TMemoryStream;
    stream := TMemoryStream.Create;
```

Inheritance and Scope

When you declare a class, you can specify its immediate ancestor. For example,

```
type TSomeControl = class(TControl);
```

declares a class called `TSomeControl` that descends from TControl. A class type automatically inherits all of the members from its immediate ancestor. Each class can declare new members and can redefine inherited ones, but a class cannot remove members defined in an ancestor. Hence `TSomeControl` contains all of the members defined in TControl and in each of TControl's ancestors.

The scope of a member's identifier starts at the point where the member is declared, continues to the end of the class declaration, and extends over all descendants of the class and the blocks of all methods defined in the class and its descendants.

TObject and TClass

The TObject class, declared in the `System` unit, is the ultimate ancestor of all other classes. TObject defines only a handful of methods, including a basic constructor and destructor. In addition to TObject, the `System` unit declares the class reference type TClass:

```
TClass = class of TObject;
```

If the declaration of a class type doesn't specify an ancestor, the class inherits directly from TObject. Thus

```
type TMyClass = class
      ...
     end;
```

is equivalent to

```
type TMyClass = class(TObject)
      ...
     end;
```

The latter form is recommended for readability.

Compatibility of Class Types

A class type is assignment-compatible with its ancestors. Hence a variable of a class type can reference an instance of any descendant type. For example, given the declarations

```
type
     TFigure = class(TObject);
     TRectangle = class(TFigure);
     TSquare = class(TRectangle);
var
     Fig: TFigure;
```

the variable `Fig` can be assigned values of type `TFigure`, `TRectangle`, and `TSquare`.

Object Types

The Win32 Delphi compiler allows an alternative syntax to class types, which you can declare object types using the syntax

```
type objectTypeName = object (ancestorObjectType)
        memberList
     end;
```

where *objectTypeName* is any valid identifier, *(ancestorObjectType)* is optional, and *memberList* declares fields, methods, and properties. If *(ancestorObjectType)* is omitted, then the new type has no ancestor. Object types cannot have published members.

Since object types do not descend from TObject, they provide no built-in constructors, destructors, or other methods. You can create instances of an object type using the `New` procedure and destroy them with the `Dispose` procedure, or you can simply declare variables of an object type, just as you would with records.

Object types are supported for backward compatibility only. Their use is not recommended on Win32, and they have been completely deprecated in the Delphi for .NET compiler.

Visibility of Class Members

Every member of a class has an attribute called visibility, which is indicated by one of the reserved words private, protected, public, published, or automated. For example,

```
published property Color: TColor read GetColor write SetColor;
```

declares a published property called `Color`. Visibility determines where and how a member can be accessed, with private representing the least accessibility, protected representing an intermediate level of accessibility, and public, published, and automated representing the greatest accessibility.

If a member's declaration appears without its own visibility specifier, the member has the same visibility as the one that precedes it. Members at the beginning of a class declaration that don't have a specified visibility are by default published, provided the class is compiled in the `{$M+}` state or is derived from a class compiled in the `{$M+}` state; otherwise, such members are public.

For readability, it is best to organize a class declaration by visibility, placing all the private members together, followed by all the protected members, and so forth. This way each visibility reserved word appears at most once and marks the beginning of a new 'section' of the declaration. So a typical class declaration should like this:

```
type
    TMyClass = class(TControl)
      private
         ...  { private declarations here }
      protected
         ...  { protected declarations here }
      public
         ...  { public declarations here }
      published
         ...  { published declarations here }
    end;
```

You can increase the visibility of a member in a descendant class by redeclaring it, but you cannot decrease its visibility. For example, a protected property can be made public in a descendant, but not private. Moreover, published members cannot become public in a descendant class. For more information, see Property overrides and redeclarations.

Private, Protected, and Public Members

A private member is invisible outside of the unit or program where its class is declared. In other words, a private method cannot be called from another module, and a private field or property cannot be read or written to from another module. By placing related class declarations in the same module, you can give the classes access to one another's private members without making those members more widely accessible.

A protected member is visible anywhere in the module where its class is declared and from any descendant class, regardless of the module where the descendant class appears. A protected method can be called, and a protected field or property read or written to, from the definition of any method belonging to a class that descends from the one where the protected member is declared. Members that are intended for use only in the implementation of derived classes are usually protected.

A public member is visible wherever its class can be referenced.

Additional Visibility Specifiers for .NET

In addition to private and protected visibility specifiers, the Delphi for .NET compiler supports additional visibility settings that comply with the .NET Common Language Specification (CLS). These are, strict private, and strict protected visibility.

Class members with strict private visibility are accessible only within the class in which they are declared. They are not visible to procedures or functions declared within the same unit.

Class members with strict protected visibility are visible within the class in which they are declared, and within any descendant class, regardless of where it is declared.

Delphi's traditional private visibility specifier maps to the CLR's `assembly` visibility. Delphi's protected visibility specifier maps to the CLR's `assembly` or `family` visibility.

Note: The word *strict* is treated as a directive within the context of a class declaration. Within a class declaration you cannot declare a member named 'strict', but it is acceptable for use outside of a class declaration.

Published Members

Published members have the same visibility as public members. The difference is that runtime type information (RTTI) is generated for published members. RTTI allows an application to query the fields and properties of an object dynamically and to locate its methods. RTTI is used to access the values of properties when saving and loading form files, to display properties in the **Object Inspector**, and to associate specific methods (called event handlers) with specific properties (called events).

Published properties are restricted to certain data types. Ordinal, string, class, interface, variant, and method-pointer types can be published. So can set types, provided the upper and lower bounds of the base type have ordinal values between 0 and 31. (In other words, the set must fit in a byte, word, or double word.) Any real type except Real48 can be published. Properties of an array type (as distinct from array properties, discussed below) cannot be published.

Some properties, although publishable, are not fully supported by the streaming system. These include properties of record types, array properties of all publishable types, and properties of enumerated types that include anonymous values. If you publish a property of this kind, the **Object Inspector** won't display it correctly, nor will the property's value be preserved when objects are streamed to disk.

All methods are publishable, but a class cannot publish two or more overloaded methods with the same name. Fields can be published only if they are of a class or interface type.

A class cannot have published members unless it is compiled in the `{$M+}` state or descends from a class compiled in the `{$M+}` state. Most classes with published members derive from TPersistent, which is compiled in the `{$M+}` state, so it is seldom necessary to use the `$M` directive.

Note: Identifiers that contain Unicode characters are not allowed in published sections of classes, or in types used by published members.

Automated Members (Win32 Only)

Automated members have the same visibility as public members. The difference is that Automation type information (required for Automation servers) is generated for automated members. Automated members typically appear only in Win32 classes, and the automated reserved word has been deprecated in the .NET compiler. The automated reserved word is maintained for backward compatibility. The TAutoObject class in the ComObj unit does not use automated.

The following restrictions apply to methods and properties declared as automated.

- The types of all properties, array property parameters, method parameters, and function results must be automatable. The automatable types are Byte, Currency, Real, Double, Longint, Integer, Single, Smallint, AnsiString, WideString, TDateTime, Variant, OleVariant, WordBool, and all interface types.
- Method declarations must use the default register calling convention. They can be virtual, but not dynamic.
- Property declarations can include access specifiers (read and write) but other specifiers (index, stored, default, and nodefault) are not allowed. Access specifiers must list a method identifier that uses the default register calling convention; field identifiers are not allowed.
- Property declarations must specify a type. Property overrides are not allowed.

The declaration of an automated method or property can include a dispid directive. Specifying an already used ID in a dispid directive causes an error.

On the Win32 platform, this directive must be followed by an integer constant that specifies an Automation dispatch ID for the member. Otherwise, the compiler automatically assigns the member a dispatch ID that is one larger than the largest dispatch ID used by any method or property in the class and its ancestors. For more information about Automation (on Win32 only), see Automation objects.

Forward Declarations and Mutually Dependent Classes

If the declaration of a class type ends with the word class and a semicolon - that is, if it has the form

type *className* = class;

with no ancestor or class members listed after the word class, then it is a forward declaration. A forward declaration must be resolved by a defining declaration of the same class within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent classes. For example,

```
type
      TFigure = class;   // forward declaration
      TDrawing = class
         Figure: TFigure;
           ...
      end;

      TFigure = class   // defining declaration
         Drawing: TDrawing;
           ...
      end;
```

Do not confuse forward declarations with complete declarations of types that derive from TObject without declaring any class members.

```
type
     TFirstClass = class;   // this is a forward declaration
     TSecondClass = class   // this is a complete class declaration
     end;                   //
     TThirdClass = class(TObject);  // this is a complete class declaration
```

Fields

This topic describes the syntax of class data fields declarations.

About Fields

A field is like a variable that belongs to an object. Fields can be of any type, including class types. (That is, fields can hold object references.) Fields are usually private.

To define a field member of a class, simply declare the field as you would a variable. All field declarations must occur before any property or method declarations. For example, the following declaration creates a class called `TNumber` whose only member, other than the methods is inherits from TObject, is an integer field called `Int`.

```
type TNumber = class
      Int: Integer;
     end;
```

Fields are statically bound; that is, references to them are fixed at compile time. To see what this means, consider the following code.

```
type
   TAncestor = class
      Value: Integer;
   end;

   TDescendant = class(TAncestor)
      Value: string;    // hides the inherited Value field
   end;

var
   MyObject: TAncestor;

begin
   MyObject := TDescendant.Create;
   MyObject.Value := 'Hello!'     // error

  (MyObject as TDescendant).Value := 'Hello!'   // works!
end;
```

Although `MyObject` holds an instance of `TDescendant`, it is declared as `TAncestor`. The compiler therefore interprets `MyObject.Value` as referring to the (integer) field declared in `TAncestor`. Both fields, however, exist in the `TDescendant` object; the inherited `Value` is hidden by the new one, and can be accessed through a typecast.

Constants, and typed constant declarations can appear in classes and non-anonymous records at global scope. Both constants and typed constants can also appear within nested type definitions. Constants and typed constants can appear only within class definitions when the class is defined locally to a procedure (i.e. they cannot appear within records defined within a procedure).

Class Fields (.NET)

Class fields are data fields in a class that can be accessed without an object reference.

You can introduce a block of class fields within a class declaration by using the class var block declaration. All fields declared after class var have static storage attributes. A class var block is terminated by the following:

1. Another class var declaration

2. A procedure or function (i.e. method) declaration (including class procedures and class functions)
3. A property declaration (including class properties)
4. A constructor or destructor declaration
5. A visibility scope specifier (public, private, protected, published, strict private, and strict protected)

For example:

```
type
    TMyClass = class
      public
        class var        // Introduce a block of class static fields.
          Red: Integer;
          Green: Integer;
          Blue: Integer;
          procedure Proc1; // Ends the class var block.
    end;
```

The above class fields can be accessed with the code:

```
TMyClass.Red := 0;
TMyClass.Green := 0;
TMyClass.Blue := 0;
```

Methods

A method is a procedure or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example, `SomeObject.Free` calls the `Free` method in `SomeObject`.

This topic covers the following material:

- Methods declarations and implementation
- Method binding
- Overloading methods
- Constructors and destructors
- Message methods

About Methods

Within a class declaration, methods appear as procedure and function headings, which work like forward declarations. Somewhere after the class declaration, but within the same module, each method must be implemented by a defining declaration. For example, suppose the declaration of `TMyClass` includes a method called `DoSomething`:

```
type
   TMyClass = class(TObject)
      ...
      procedure DoSomething;
      ...
   end;
```

A defining declaration for `DoSomething` must occur later in the module:

```
procedure TMyClass.DoSomething;
begin
     ...
end;
```

While a class can be declared in either the interface or the implementation section of a unit, defining declarations for a class' methods must be in the implementation section.

In the heading of a defining declaration, the method name is always qualified with the name of the class to which it belongs. The heading can repeat the parameter list from the class declaration; if it does, the order, type and names of the parameters must match exactly, and if the method is a function, the return value must match as well.

Method declarations can include special directives that are not used with other functions or procedures. Directives should appear in the class declaration only, not in the defining declaration, and should always be listed in the following order:

reintroduce; overload; *binding; calling convention;*abstract; *warning*

where *binding* is virtual, dynamic, or override; calling convention is register, pascal, cdecl, stdcall, or safecall; and *warning* is platform, deprecated, or library.

Inherited

The reserved word inherited plays a special role in implementing polymorphic behavior. It can occur in method definitions, with or without an identifier after it.

If inherited is followed by the name of a member, it represents a normal method call or reference to a property or field - except that the search for the referenced member begins with the immediate ancestor of the enclosing method's class. For example, when

```
inherited Create(...);
```

occurs in the definition of a method, it calls the inherited `Create`.

When inherited has no identifier after it, it refers to the inherited method with the same name as the enclosing method or, if the enclosing method is a message handler, to the inherited message handler for the same message. In this case, inherited takes no explicit parameters, but passes to the inherited method the same parameters with which the enclosing method was called. For example,

```
inherited;
```

occurs frequently in the implementation of constructors. It calls the inherited constructor with the same parameters that were passed to the descendant.

Self

Within the implementation of a method, the identifier Self references the object in which the method is called. For example, here is the implementation of TCollection's `Add` method in the `Classes` unit.

```
function TCollection.Add: TCollectionItem;
begin
    Result := FItemClass.Create(Self);
end;
```

The `Add` method calls the `Create` method in the class referenced by the `FItemClass` field, which is always a TCollectionItem descendant. `TCollectionItem.Create` takes a single parameter of type TCollection, so `Add` passes it the TCollection instance object where `Add` is called. This is illustrated in the following code.

```
var MyCollection: TCollection;
    ...
    MyCollection.Add   // MyCollection is passed to the TCollectionItem.Create method
```

Self is useful for a variety of reasons. For example, a member identifier declared in a class type might be redeclared in the block of one of the class' methods. In this case, you can access the original member identifier as `Self.Identifier`.

For information about Self in class methods, see Class methods.

Method Binding

Method bindings can be static (the default), virtual, or dynamic. Virtual and dynamic methods can be overridden, and they can be abstract. These designations come into play when a variable of one class type holds a value of a descendant class type. They determine which implementation is activated when a method is called.

Static Methods

Methods are by default static. When a static method is called, the declared (compile-time) type of the class or object variable used in the method call determines which implementation to activate. In the following example, the `Draw` methods are static.

```
type
    TFigure = class
      procedure Draw;
    end;

    TRectangle = class(TFigure)
      procedure Draw;
    end;
```

Given these declarations, the following code illustrates the effect of calling a static method. In the second call to `Figure.Draw`, the `Figure` variable references an object of class `TRectangle`, but the call invokes the implementation of `Draw` in `TFigure`, because the declared type of the `Figure` variable is `TFigure`.

```
var
    Figure: TFigure;
    Rectangle: TRectangle;

    begin
            Figure := TFigure.Create;
            Figure.Draw;              // calls TFigure.Draw
            Figure.Destroy;
            Figure := TRectangle.Create;
            Figure.Draw;              // calls TFigure.Draw

            TRectangle(Figure).Draw;  // calls TRectangle.Draw

            Figure.Destroy;
            Rectangle := TRectangle.Create;
            Rectangle.Draw;          // calls TRectangle.Draw
            Rectangle.Destroy;
    end;
```

Virtual and Dynamic Methods

To make a method virtual or dynamic, include the virtual or dynamic directive in its declaration. Virtual and dynamic methods, unlike static methods, can be overridden in descendant classes. When an overridden method is called, the actual (runtime) type of the class or object used in the method callnot the declared type of the variabledetermines which implementation to activate.

To override a method, redeclare it with the override directive. An override declaration must match the ancestor declaration in the order and type of its parameters and in its result type (if any).

In the following example, the `Draw` method declared in `TFigure` is overridden in two descendant classes.

```
type
    TFigure = class
      procedure Draw; virtual;
    end;
```

```
  TRectangle = class(TFigure)
    procedure Draw; override;
  end;

  TEllipse = class(TFigure)
    procedure Draw; override;
  end;
```

Given these declarations, the following code illustrates the effect of calling a virtual method through a variable whose actual type varies at runtime.

```
var
   Figure: TFigure;

   begin
     Figure := TRectangle.Create;
     Figure.Draw;      // calls TRectangle.Draw
     Figure.Destroy;
     Figure := TEllipse.Create;
     Figure.Draw;      // calls TEllipse.Draw
     Figure.Destroy;
   end;
```

Only virtual and dynamic methods can be overridden. All methods, however, can be overloaded; see Overloading methods.

The Delphi for .NET compiler supports the concept of a *final* virtual method. When the keyword final is applied to a virtual method, no ancestor class can override that method. Use of the final keyword is an important design decision that can help document how the class is intended to be used. It can also give the .NET JIT compiler hints that allow it to optimize the code it produces.

Virtual Versus Dynamic

Virtual and dynamic methods are semantically equivalent. They differ only in the implementation of method-call dispatching at runtime. Virtual methods optimize for speed, while dynamic methods optimize for code size.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful when a base class declares many overridable methods which are inherited by many descendant classes in an application, but only occasionally overridden.

Note: Only use dynamic methods if there is a clear, observable benefit. Generally, use virtual methods.

Overriding Versus Hiding

If a method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn't include override, the new declaration merely hides the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound. For example,

```
type
   T1 = class(TObject)
      procedure Act; virtual;
   end;

   T2 = class(T1)
      procedure Act;   // Act is redeclared, but not overridden
   end;
```

```
var
   SomeObject: T1;

begin
   SomeObject := T2.Create;
   SomeObject.Act;    // calls T1.Act
end;
```

Reintroduce

The reintroduce directive suppresses compiler warnings about hiding previously declared virtual methods. For example,

```
procedure DoSomething; reintroduce;   // the ancestor class also has a DoSomething method
```

Use reintroduce when you want to hide an inherited virtual method with a new one.

Abstract Methods

An abstract method is a virtual or dynamic method that has no implementation in the class where it is declared. Its implementation is deferred to a descendant class. Abstract methods must be declared with the directive abstract after virtual or dynamic. For example,

```
procedure DoSomething; virtual; abstract;
```

You can call an abstract method only in a class or instance of a class in which the method has been overridden.

Note: The Delphi for .NET compiler allows an entire class to be declared abstract, even though it does not contain any virtual abstract methods. See Class Types for more information.

Class Static Methods (.NET)

Like class fields and class properties, class static methods can be accessed without an object reference. Unlike Win32 Delphi class methods, class static methods have no Self parameter at all, and therefore cannot access any class members. Also unlike Win32 Delphi, class static methods cannot be declared virtual.

Methods are made class static by appending the word static to their declaration, for example

```
type
   TMyClass = class
     strict private
       class var
       FX: Integer;

     strict protected

       // Note: accessors for class properties must be declared class static.
       class function GetX: Integer; static;
       class procedure SetX(val: Integer); static;

     public
       class property X: Integer read GetX write SetX;
       class procedure StatProc(s: String); static;
   end;
```

You can call a class static method through the class type (i.e. without having an object reference), for example

```
TMyClass.X := 17;
TMyClass.StatProc('Hello');
```

Overloading Methods

A method can be redeclared using the overload directive. In this case, if the redeclared method has a different parameter signature from its ancestor, it overloads the inherited method without hiding it. Calling the method in a descendant class activates whichever implementation matches the parameters in the call.

If you overload a virtual method, use the reintroduce directive when you redeclare it in descendant classes. For example,

```
type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
  end;

  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
  ...

SomeObject := T2.Create;
SomeObject.Test('Hello!');       // calls T2.Test
SomeObject.Test(7);              // calls T1.Test
```

Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.

```
type
    TSomeClass = class
      published
```

```
        function Func(P: Integer): Integer;
        function Func(P: Boolean): Integer;   // error
          ...
```

Methods that serve as property read or write specifiers cannot be overloaded.

The implementation of an overloaded method must repeat the parameter list from the class declaration. For more information about overloading, see Overloading procedures and functions.

Constructors

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a procedure declaration, but it begins with the word constructor. Examples:

```
constructor Create;
constructor Create(AOwner: TComponent);
```

Constructors must use the default register calling convention. Although the declaration specifies no return value, a constructor returns a reference to the object it creates or is called in.

A class can have more than one constructor, but most have only one. It is conventional to call the constructor `Create`.

To create an object, call the constructor method on a class type. For example,

```
MyObject := TMyClass.Create;
```

This allocates storage for the new object, sets the values of all ordinal fields to zero, assigns nil to all pointer and class-type fields, and makes all string fields empty. Other actions specified in the constructor implementation are performed next; typically, objects are initialized based on values passed as parameters to the constructor. Finally, the constructor returns a reference to the newly allocated and initialized object. The type of the returned value is the same as the class type specified in the constructor call.

If an exception is raised during execution of a constructor that was invoked on a class reference, the `Destroy` destructor is automatically called to destroy the unfinished object.

When a constructor is called using an object reference (rather than a class reference), it does not create an object. Instead, the constructor operates on the specified object, executing only the statements in the constructor's implementation, and then returns a reference to the object. A constructor is typically invoked on an object reference in conjunction with the reserved word inherited to execute an inherited constructor.

Here is an example of a class type and its constructor.

```
type
   TShape = class(TGraphicControl)
     private
       FPen: TPen;
       FBrush: TBrush;
       procedure PenChanged(Sender: TObject);
       procedure BrushChanged(Sender: TObject);
     public
       constructor Create(Owner: TComponent); override;
       destructor Destroy; override;
       ...
   end;

constructor TShape.Create(Owner: TComponent);
```

```
begin
    inherited Create(Owner);     // Initialize inherited parts
    Width := 65;          // Change inherited properties
    Height := 65;
    FPen := TPen.Create;  // Initialize new fields
    FPen.OnChange := PenChanged;
    FBrush := TBrush.Create;
    FBrush.OnChange := BrushChanged;
end;
```

The first action of a constructor is usually to call an inherited constructor to initialize the object's inherited fields. The constructor then initializes the fields introduced in the descendant class. Because a constructor always clears the storage it allocates for a new object, all fields start with a value of zero (ordinal types), nil (pointer and class types), empty (string types), or Unassigned (variants). Hence there is no need to initialize fields in a constructor's implementation except to nonzero or nonempty values.

When invoked through a class-type identifier, a constructor declared as virtual is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objectsthat is, construction of objects whose types aren't known at compile time. (See Class references.)

Note: For more information on constructors, destructors, and memory management issues on the .NET platform, please see the topic Memory Management Issues on the .NET Platform.

The Class Constructor (.NET)

A class constructor executes before a class is referenced or used. The class constructor must be declared as strict private, and there can be at most one class constructor declared in a class. Descendants can declare their own class constructor, however, do not call inherited within the body of a class constructor. In fact, you cannot call a class constructor directly, or access it in any way (such as taking its address). The compiler generates code to call class constructors for you.

There can be no guarantees on when a class constructor will execute, except to say that it will execute at some time before the class is used. On the .NET platform in order for a class to be "used", it must reside in code that is actually executed. For example, if a class is first referenced in an if statement, and the test of the if statement is never true during the course of execution, then the class will never be loaded and JIT compiled. Hence, in this case the class constructor would not be called.

The following class declaration demonstrates the syntax of class properties and fields, as well as class static methods and class constructors:

```
type
    TMyClass = class
      strict protected

        // Accessors for class properties must be declared class static.
        class function GetX: Integer; static;
        class procedure SetX(val: Integer); static;
      public
        class property X: Integer read GetX write SetX;
        class procedure StatProc(s: String); static;
      strict private
        class var
          FX: Integer;
          class constructor Create;
    end;
```

Destructors

A destructor is a special method that destroys the object where it is called and deallocates its memory. The declaration of a destructor looks like a procedure declaration, but it begins with the word destructor. Example:

```
destructor SpecialDestructor(SaveData: Boolean);
destructor Destroy; override;
```

Destructors on Win32 must use the default register calling convention. Although a class can have more than one destructor, it is recommended that each class override the inherited `Destroy` method and declare no other destructors.

To call a destructor, you must reference an instance object. For example,

```
MyObject.Destroy;
```

When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

Here is an example of a destructor implementation.

```
destructor TShape.Destroy;
begin
    FBrush.Free;
    FPen.Free;
    inherited Destroy;
end;
```

The last action in a destructor's implementation is typically to call the inherited destructor to destroy the object's inherited fields.

When an exception is raised during creation of an object, Destroy is automatically called to dispose of the unfinished object. This means that `Destroy` must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and pointer-type fields in a partially constructed object are always nil. A destructor should therefore check for nil values before operating on class-type or pointer-type fields. Calling the Free method (defined in TObject), rather than `Destroy`, offers a convenient way of checking for nil values before destroying an object.

Note: For more information on constructors, destructors, and memory management issues on the .NET platform, please see the topic Memory Management Issues on the .NET Platform.

Message Methods

Message methods implement responses to dynamically dispatched messages. The message method syntax is supported on all platforms. VCL uses message methods to respond to Windows messages.

A message method is created by including the message directive in a method declaration, followed by an integer constant between 1 and 49151 which specifies the message ID. For message methods in VCL controls, the integer constant can be one of the Win32 message IDs defined, along with corresponding record types, in the `Messages` unit. A message method must be a procedure that takes a single var parameter.

For example:

```
type
    TTextBox = class(TCustomControl)
      private
       procedure WMChar(var Message: TWMChar); message WM_CHAR;
       ...
    end;
```

A message method does not have to include the override directive to override an inherited message method. In fact, it doesn't have to specify the same method name or parameter type as the method it overrides. The message ID alone determines which message the method responds to and whether it is an override.

Implementing Message Methods

The implementation of a message method can call the inherited message method, as in this example:

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
   if Message.CharCode = Ord(#13) then
      ProcessEnter
   else
      inherited;
end;
```

The inherited statement searches backward through the class hierarchy and invokes the first message method with the same ID as the current method, automatically passing the message record to it. If no ancestor class implements a message method for the given ID, inherited calls the `DefaultHandler` method originally defined in TObject.

The implementation of `DefaultHandler` in TObject simply returns without performing any actions. By overriding `DefaultHandler`, a class can implement its own default handling of messages. On Win32, the `DefaultHandler` method for controls calls the Win32 API `DefWindowProc`.

Message Dispatching

Message handlers are seldom called directly. Instead, messages are dispatched to an object using the `Dispatch` method inherited from TObject:

```
procedure Dispatch(var Message);
```

The `Message` parameter passed to `Dispatch` must be a record whose first entry is a field of type Word containing a message ID.

`Dispatch` searches backward through the class hierarchy (starting from the class of the object where it is called) and invokes the first message method for the ID passed to it. If no message method is found for the given ID, `Dispatch` calls `DefaultHandler`.

Properties

This topic describes the following material:

- Property access
- Array properties
- Index specifiers
- Storage specifiers
- Property overrides and redeclarations
- Class properties (.NET)

About Properties

A property, like a field, defines an attribute of an object. But while a field is merely a storage location whose contents can be examined and changed, a property associates specific actions with reading or modifying its data. Properties provide control over access to an object's attributes, and they allow attributes to be computed.

The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is

property *propertyName[indexes]: type* `index` *integerConstant specifiers;*

where

- *propertyName* is any valid identifier.
- *[indexes]* is optional and is a sequence of parameter declarations separated by semicolons. Each parameter declaration has the form *identifier1*, ..., *identifiern*: type. For more information, see Array Properties, below.
- type must be a predefined or previously declared type identifier. That is, property declarations like `property Num: 0..9` ... are invalid.
- the index *integerConstant* clause is optional. For more information, see Index Specifiers, below.
- specifiers is a sequence of read, write, stored, default (or nodefault), and implements specifiers. Every property declaration must have at least one read or write specifier.

Properties are defined by their access specifiers. Unlike fields, properties cannot be passed as var parameters, nor can the **@** operator be applied to a property. The reason is that a property doesn't necessarily exist in memory. It could, for instance, have a read method that retrieves a value from a database or generates a random value.

Property Access

Every property has a read specifier, a write specifier, or both. These are called access specifiers and they have the form

read *fieldOrMethod*

write *fieldOrMethod*

where *fieldOrMethod* is the name of a field or method declared in the same class as the property or in an ancestor class.

- If *fieldOrMethod* is declared in the same class, it must occur before the property declaration. If it is declared in an ancestor class, it must be visible from the descendant; that is, it cannot be a private field or method of an ancestor class declared in a different unit.
- If *fieldOrMethod* is a field, it must be of the same type as the property.

- If *fieldOrMethod* is a method, it cannot be dynamic and, if virtual, cannot be overloaded. Moreover, access methods for a published property must use the default register calling convention.
- In a read specifier, if fieldOrMethod is a method, it must be a parameterless function whose result type is the same as the property's type. (An exception is the access method for an indexed property or an array property.)
- In a write specifier, if *fieldOrMethod* is a method, it must be a procedure that takes a single value or const parameter of the same type as the property (or more, if it is an array property or indexed property).

For example, given the declaration

```
property Color: TColor read GetColor write SetColor;
```

the `GetColor` method must be declared as

```
function GetColor: TColor;
```

and the `SetColor` method must be declared as one of these:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

(The name of `SetColor`'s parameter, of course, doesn't have to be `Value`.)

When a property is referenced in an expression, its value is read using the field or method listed in the read specifier. When a property is referenced in an assignment statement, its value is written using the field or method listed in the write specifier.

The example below declares a class called `TCompass` with a published property called `Heading`. The value of `Heading` is read through the `FHeading` field and written through the `SetHeading` procedure.

```
type
   THeading = 0..359;
   TCompass = class(TControl)
     private
        FHeading: THeading;
        procedure SetHeading(Value: THeading);
     published
        property Heading: THeading read FHeading write SetHeading;
        ...
    end;
```

Given this declaration, the statements

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

correspond to

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

In the `TCompass` class, no action is associated with reading the `Heading` property; the read operation consists of retrieving the value stored in the `FHeading` field. On the other hand, assigning a value to the `Heading` property translates into a call to the `SetHeading` method, which, presumably, stores the new value in the `FHeading` field as well as performing other actions. For example, `SetHeading` might be implemented like this:

```
procedure TCompass.SetHeading(Value: THeading);
begin
            if FHeading <> Value then
             begin
               FHeading := Value;
               Repaint;    // update user interface to reflect new value
             end;
end;
```

A property whose declaration includes only a read specifier is a read-only property, and one whose declaration includes only a write specifier is a write-only property. It is an error to assign a value to a read-only property or use a write-only property in an expression.

Array Properties

Array properties are indexed properties. They can represent things like items in a list, child controls of a control, and pixels of a bitmap.

The declaration of an array property includes a parameter list that specifies the names and types of the indexes. For example,

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

The format of an index parameter list is the same as that of a procedure's or function's parameter list, except that the parameter declarations are enclosed in brackets instead of parentheses. Unlike arrays, which can use only ordinal-type indexes, array properties allow indexes of any type.

For array properties, access specifiers must list methods rather than fields. The method in a read specifier must be a function that takes the number and type of parameters listed in the property's index parameter list, in the same order, and whose result type is identical to the property's type. The method in a write specifier must be a procedure that takes the number and type of parameters listed in the property's index parameter list, in the same order, plus an additional value or const parameter of the same type as the property.

For example, the access methods for the array properties above might be declared as

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

An array property is accessed by indexing the property identifier. For example, the statements

```
if Collection.Objects[0] = nil then Exit;
```

```
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\BIN';
```

correspond to

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\BIN');
```

The definition of an array property can be followed by the default directive, in which case the array property becomes the default property of the class. For example,

```
type
   TStringArray = class
    public
       property Strings[Index: Integer]: string ...; default;
          ...
    end;
```

If a class has a default property, you can access that property with the abbreviation `object[index]`, which is equivalent to `object.property[index]`. For example, given the declaration above, `StringArray.Strings[7]` can be abbreviated to `StringArray[7]`. A class can have only one default property. Changing or hiding the default property in descendant classes may lead to unexpected behavior, since the compiler always binds to properties statically.

Index Specifiers

Index specifiers allow several properties to share the same access method while representing different values. An index specifier consists of the directive index followed by an integer constant between -2147483647 and 2147483647. If a property has an index specifier, its read and write specifiers must list methods rather than fields. For example,

```
type
   TRectangle = class
     private
       FCoordinates: array[0..3] of Longint;
       function GetCoordinate(Index: Integer): Longint;
       procedure SetCoordinate(Index: Integer; Value: Longint);
     public
       property Left: Longint index 0 read GetCoordinate write SetCoordinate;
       property Top: Longint index 1 read GetCoordinate write SetCoordinate;
       property Right: Longint index 2 read GetCoordinate write SetCoordinate;
       property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
      property Coordinates[Index: Integer]: Longint read GetCoordinate write SetCoordinate;
       ...
   end;
```

An access method for a property with an index specifier must take an extra value parameter of type Integer. For a read function, it must be the last parameter; for a write procedure, it must be the second-to-last parameter (preceding the parameter that specifies the property value). When a program accesses the property, the property's integer constant is automatically passed to the access method.

Given the declaration above, if `Rectangle` is of type `TRectangle`, then

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage Specifiers

The optional stored, default, and nodefault directives are called storage specifiers. They have no effect on program behavior, but control whether or not to save the values of published properties in form files.

The stored directive must be followed by True, False, the name of a Boolean field, or the name of a parameterless method that returns a Boolean value. For example,

```
property Name: TComponentName read FName write SetName stored False;
```

If a property has no stored directive, it is treated as if stored True were specified.

The default directive must be followed by a constant of the same type as the property. For example,

```
property Tag: Longint read FTag write FTag default 0;
```

To override an inherited default value without specifying a new one, use the nodefault directive. The default and nodefault directives are supported only for ordinal types and for set types, provided the upper and lower bounds of the set's base type have ordinal values between 0 and 31; if such a property is declared without default or nodefault, it is treated as if nodefault were specified. For reals, pointers, and strings, there is an implicit default value of 0, nil, and `''` (the empty string), respectively.

Note: You can't use the ordinal value 2147483648 has a default value. This value is used internally to represent nodefault.

When saving a component's state, the storage specifiers of the component's published properties are checked. If a property's current value is different from its default value (or if there is no default value) and the stored specifier is True, then the property's value is saved. Otherwise, the property's value is not saved.

Note: Property values are not automatically initialized to the default value. That is, the default directive controls only when property values are saved to the form file, but not the initial value of the property on a newly created instance.

Storage specifiers are not supported for array properties. The default directive has a different meaning when used in an array property declaration. See Array Properties, above.

Property Overrides and Redeclarations

A property declaration that doesn't specify a type is called a property override. Property overrides allow you to change a property's inherited visibility or specifiers. The simplest override consists only of the reserved word property followed by an inherited property identifier; this form is used to change a property's visibility. For example, if an ancestor class declares a property as protected, a derived class can redeclare it in a public or published section of the class. Property overrides can include read, write,stored, default, and nodefault directives; any such directive overrides the corresponding inherited directive. An override can replace an inherited access specifier, add a missing specifier, or increase a property's visibility, but it cannot remove an access specifier or decrease a property's visibility. An override can include an implements directive, which adds to the list of implemented interfaces without removing inherited ones.

The following declarations illustrate the use of property overrides.

```
type
   TAncestor = class
       ...
     protected
       property Size: Integer read FSize;
       property Text: string read GetText write SetText;
       property Color: TColor read FColor write SetColor stored False;
       ...
   end;

type

   TDerived = class(TAncestor)
       ...
     protected
       property Size write SetSize;
     published
       property Text;
       property Color stored True default clBlue;
       ...
   end;
```

The override of `Size` adds a write specifier to allow the property to be modified. The overrides of `Text` and `Color` change the visibility of the properties from protected to published. The property override of `Color` also specifies that the property should be filed if its value isn't `clBlue`.

A redeclaration of a property that includes a type identifier hides the inherited property rather than overriding it. This means that a new property is created with the same name as the inherited one. Any property declaration that specifies a type must be a complete declaration, and must therefore include at least one access specifier.

Whether a property is hidden or overridden in a derived class, property look-up is always static. That is, the declared (compile-time) type of the variable used to identify an object determines the interpretation of its property identifiers. Hence, after the following code executes, reading or assigning a value to `MyObject.Value` invokes `Method1` or `Method2`, even though `MyObject` holds an instance of `TDescendant`. But you can cast `MyObject` to `TDescendant` to access the descendant class's properties and their access specifiers.

```
type
    TAncestor = class
      ...
      property Value: Integer read Method1 write Method2;
    end;

    TDescendant = class(TAncestor)
      ...
      property Value: Integer read Method3 write Method4;
    end;

 var MyObject: TAncestor;
      ...
     MyObject := TDescendant.Create;
```

Class Properties (.NET)

Class properties can be accessed without an object reference. Class property accessors must themselves be declared as class static methods, or class fields. A class property cannot be published, and cannot have stored or default value definitions.

You can introduce a block of class static properties within a class declaration by using the class var block declaration. All properties declared after class var have static storage attributes. A class var block is terminated by the following:

1. Another class var declaration
2. A procedure or function (i.e. method) declaration (including class procedures and class functions)
3. A property declaration (including class properties)
4. A constructor or destructor declaration
5. A visibility scope specifier (public, private, protected, published, strict private, and strict protected)

For example:

```
type
   TMyClass = class
     strict private
       class var         // Note fields must be declared as class fields
          FRed: Integer;
          FGreen: Integer;
          FBlue: Integer;
     public
          class var        // Introduce a block of class properties
          property Red: Integer read FRed write FRed;
          Green: Integer read FGreen write FGreen;
          Blue: Integer read FBlue write FBlue;
          procedure Proc1; // Ends the class var block.
   end;
```

You can access the above class properites with the code:

```
TMyClass.Red := 0;
TMyClass.Blue := 0;
TMyClass.Green := 0;
```

Class References

Sometimes operations are performed on a class itself, rather than on instances of a class (that is, objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*.

This topic covers the following material:

- Class reference types
- Class operators
- Class methods

Class-Reference Types

A class-reference type, sometimes called a metaclass, is denoted by a construction of the form

`class of` *type*

where *type* is any class type. The identifier *type* itself denotes a value whose type is `class of` *type*. If `type1` is an ancestor of `type2`, then `class of type2` is assignment-compatible with class of `type1`. Thus

```
type TClass = class of TObject;
var AnyObj: TClass;
```

declares a variable called `AnyObj` that can hold a reference to any class. (The definition of a class-reference type cannot occur directly in a variable declaration or parameter list.) You can assign the value nil to a variable of any class-reference type.

To see how class-reference types are used, look at the declaration of the constructor for TCollection (in the `Classes` unit):

```
type TCollectionItemClass = class of TCollectionItem;
     ...
constructor Create(ItemClass: TCollectionItemClass);
```

This declaration says that to create a TCollection instance object, you must pass to the constructor the name of a class descending from TCollectionItem.

Class-reference types are useful when you want to invoke a class method or virtual constructor on a class or object whose actual type is unknown at compile time.

Constructors and Class References

A constructor can be called using a variable of a class-reference type. This allows construction of objects whose type isn't known at compile time. For example,

```
type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
const ControlName: string; X, Y, W, H: Integer): TControl;
begin
```

```
      Result := ControlClass.Create(MainForm);
      with Result do
       begin
         Parent := MainForm;
         Name := ControlName;
         SetBounds(X, Y, W, H);
         Visible := True;
       end;
end;
```

The `CreateControl` function requires a class-reference parameter to tell it what kind of control to create. It uses this parameter to call the class's constructor. Because class-type identifiers denote class-reference values, a call to `CreateControl` can specify the identifier of the class to create an instance of. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Constructors called using class references are usually virtual. The constructor implementation activated by the call depends on the runtime type of the class reference.

Class Operators

Every class inherits from TObject methods called ClassType and ClassParent that return, respectively, a reference to the class of an object and of an object's immediate ancestor. Both methods return a value of type TClass (where `TClass = class of TObject`), which can be cast to a more specific type. Every class also inherits a method called InheritsFrom that tests whether the object where it is called descends from a specified class. These methods are used by the is and as operators, and it is seldom necessary to call them directly.

The is Operator

The is operator, which performs dynamic type checking, is used to verify the actual runtime class of an object. The expression

*object*is*class*

returns True if *object* is an instance of the class denoted by class or one of its descendants, and False otherwise. (If object is nil, the result is False.) If the declared type of object is unrelated to class - that is, if the types are distinct and one is not an ancestor of the othera compilation error results. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

This statement casts a variable to `TEdit` after first verifying that the object it references is an instance of `TEdit` or one of its descendants.

The as Operator

The as operator performs checked typecasts. The expression

*object*as*class*

returns a reference to the same object as object, but with the type given by class. At runtime, object must be an instance of the class denoted by class or one of its descendants, or be nil; otherwise an exception is raised. If the declared type of object is unrelated to class - that is, if the types are distinct and one is not an ancestor of the other - a compilation error results. For example,

```
with Sender as TButton do
 begin
  Caption := '&Ok';
  OnClick := OkClick;
 end;
```

The rules of operator precedence often require as typecasts to be enclosed in parentheses. For example,

```
(Sender as TButton).Caption := '&Ok';
```

Class Methods

A class method is a method (other than a constructor) that operates on classes instead of objects. The definition of a class method must begin with the reserved word class. For example,

```
type
  TFigure = class
  public
     class function Supports(Operation: string): Boolean; virtual;
     class procedure GetInfo(var Info: TFigureInfo); virtual;
     ...
  end;
```

The defining declaration of a class method must also begin with class. For example,

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
    ...
end;
```

In the defining declaration of a class method, the identifier Self represents the class where the method is called (which could be a descendant of the class in which it is defined). If the method is called in the class `C`, then Self is of the type class of `C`. Thus you cannot use Self to access fields, properties, and normal (object) methods, but you can use it to call constructors and other class methods.

A class method can be called through a class reference or an object reference. When it is called through an object reference, the class of the object becomes the value of Self.

Exceptions

This topic covers the following material:

- A conceptual overview of exceptions and exception handling
- Declaring exception types
- Raising and handling exceptions

About Exceptions

An exception is raised when an error or other event interrupts normal execution of a program. The exception transfers control to an exception handler, which allows you to separate normal program logic from error-handling. Because exceptions are objects, they can be grouped into hierarchies using inheritance, and new exceptions can be introduced without affecting existing code. An exception can carry information, such as an error message, from the point where it is raised to the point where it is handled.

When an application uses the `SysUtils` unit, most runtime errors are automatically converted into exceptions. Many errors that would otherwise terminate an application - such as insufficient memory, division by zero, and general protection faults - can be caught and handled.

When To Use Exceptions

Exceptions provide an elegant way to trap runtime errors without halting the program and without awkward conditional statements. The requirements imposed by exception handling semantics impose a code/data size and runtime performance penalty. While it is possible to raise exceptions for almost any reason, and to protect almost any block of code by wrapping it in a `try...except` or `try...finally` statement, in practice these tools are best reserved for special situations.

Exception handling is appropriate for errors whose chances of occurring are low or difficult to assess, but whose consequences are likely to be catastrophic (such as crashing the application); for error conditions that are complicated or difficult to test for in `if...then` statements; and when you need to respond to exceptions raised by the operating system or by routines whose source code you don't control. Exceptions are commonly used for hardware, memory, I/O, and operating-system errors.

Conditional statements are often the best way to test for errors. For example, suppose you want to make sure that a file exists before trying to open it. You could do it this way:

```
try
    AssignFile(F, FileName);
    Reset(F);     // raises an EInOutError exception if file is not found
except
    on Exception do ...
end;
```

But you could also avoid the overhead of exception handling by using

```
if FileExists(FileName) then    // returns False if file is not found; raises no exception
```

```
begin
    AssignFile(F, FileName);
```

```
    Reset(F);
end;
```

Assertions provide another way of testing a Boolean condition anywhere in your source code. When an `Assert` statement fails, the program either halts with a runtime error or (if it uses the `SysUtils` unit) raises an EAssertionFailed exception. Assertions should be used only to test for conditions that you do not expect to occur.

Declaring Exception Types

Exception types are declared just like other classes. In fact, it is possible to use an instance of any class as an exception, but it is recommended that exceptions be derived from the Exception class defined in `SysUtils`.

You can group exceptions into families using inheritance. For example, the following declarations in `SysUtils` define a family of exception types for math errors.

```
type
   EMathError = class(Exception);
   EInvalidOp = class(EMathError);
   EZeroDivide = class(EMathError);
   EOverflow = class(EMathError);
   EUnderflow = class(EMathError);
```

Given these declarations, you can define a single EMathError exception handler that also handles EInvalidOp, EZeroDivide, EOverflow, and EUnderflow.

Exception classes sometimes define fields, methods, or properties that convey additional information about the error. For example,

```
type EInOutError = class(Exception)
       ErrorCode: Integer;
     end;
```

Raising and Handling Exceptions

To raise an exception object, use an instance of the exception class with a raise statement. For example,

```
raise EMathError.Create;
```

In general, the form of a raise statement is

raise*object*at*address*

where object and at address are both optional. When an address is specified, it can be any expression that evaluates to a pointer type, but is usually a pointer to a procedure or function. For example:

```
raise Exception.Create('Missing parameter') at @MyFunction;
```

Use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred.

When an exception is raised - that is, referenced in a raise statement - it is governed by special exception-handling logic. A raise statement never returns control in the normal way. Instead, it transfers control to the innermost exception handler that can handle exceptions of the given class. (The innermost handler is the one whose `try... except` block was most recently entered but has not yet exited.)

For example, the function below converts a string to an integer, raising an ERangeError exception if the resulting value is outside a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
    Result := StrToInt(S);   // StrToInt is declared in SysUtils
    if (Result < Min) or (Result > Max) then
       raise ERangeError.CreateFmt('%d is not within the valid range of %d..%d', [Result,
Min, Max]);
end;
```

Notice the *CreateFmt* method called in the raise statement. Exception and its descendants have special constructors that provide alternative ways to create exception messages and context IDs.

A raised exception is destroyed automatically after it is handled. Never attempt to destroy a raised exception manually.

Note: Raising an exception in the initialization section of a unit may not produce the intended result. Normal exception support comes from the `SysUtils` unit, which must be initialized before such support is available. If an exception occurs during initialization, all initialized units - including `SysUtils` - are finalized and the exception is re-raised. Then the exception is caught and handled, usually by interrupting the program. Similarly, raising an exception in the finalization section of a unit may not lead to the intended result if `SysUtils` has already been finalized when the exception has been raised.

Try...except Statements

Exceptions are handled within `try...except` statements. For example,

```
try
   X := Y/Z;
   except
     on EZeroDivide do HandleZeroDivide;
end;
```

This statement attempts to divide `Y` by `Z`, but calls a routine named `HandleZeroDivide` if an EZeroDivide exception is raised.

The syntax of a `try...except` statement is

try *statements*except*exceptionBlock*end

where statements is a sequence of statements (delimited by semicolons) and *exceptionBlock* is either

- another sequence of statements or
- a sequence of exception handlers, optionally followed by

else*statements*

An exception handler has the form

on*identifier: type*do*statement*

where *identifier:* is optional (if included, identifier can be any valid identifier), type is a type used to represent exceptions, and *statement* is any statement.

A `try...except` statement executes the statements in the initial statements list. If no exceptions are raised, the exception block (*exceptionBlock*) is ignored and control passes to the next part of the program.

If an exception is raised during execution of the initial statements list, either by a raise statement in the statements list or by a procedure or function called from the statements list, an attempt is made to 'handle' the exception:

- If any of the handlers in the exception block matches the exception, control passes to the first such handler. An exception handler 'matches' an exception just in case the type in the handler is the class of the exception or an ancestor of that class.
- If no such handler is found, control passes to the statement in the else clause, if there is one.
- If the exception block is just a sequence of statements without any exception handlers, control passes to the first statement in the list.

If none of the conditions above is satisfied, the search continues in the exception block of the next-most-recently entered `try...except` statement that has not yet exited. If no appropriate handler, else clause, or statement list is found there, the search propagates to the next-most-recently entered `try...except` statement, and so forth. If the outermost `try...except` statement is reached and the exception is still not handled, the program terminates.

When an exception is handled, the stack is traced back to the procedure or function containing the `try...except` statement where the handling occurs, and control is transferred to the executed exception handler, else clause, or statement list. This process discards all procedure and function calls that occurred after entering the `try...except` statement where the exception is handled. The exception object is then automatically destroyed through a call to its `Destroy` destructor and control is passed to the statement following the `try...except` statement. (If a call to the `Exit`, `Break`, or `Continue` standard procedure causes control to leave the exception handler, the exception object is still automatically destroyed.)

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. EMathError appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

An exception handler can specify an identifier before the name of the exception class. This declares the identifier to represent the exception object during execution of the statement that follows `on...do`. The scope of the identifier is limited to that statement. For example,

```
try
  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

If the exception block specifies an else clause, the else clause handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
```

```
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

Here, the else clause handles any exception that isn't an EMathError.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```
try
   ...
except
   HandleException;
end;
```

Here, the `HandleException` routine handles any exception that occurs as a result of executing the statements between try and except.

Re-raising Exceptions

When the reserved word raise occurs in an exception block without an object reference following it, it raises whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way and then re-raise the exception. Re-raising is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

For example, the `GetFileList` function allocates a `TStringList` object and fills it with file names matching a specified search path:

```
function GetFileList(const Path: string): TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
    while I = 0 do
      begin
          Result.Add(SearchRec.Name);
          I := FindNext(SearchRec);
      end;
  except
      Result.Free;
      raise;
  end;
end;
```

`GetFileList` creates a `TStringList` object, then uses the `FindFirst` and `FindNext` functions (defined in `SysUtils`) to initialize it. If the initialization fails - for example because the search path is invalid, or because there is not enough memory to fill in the string list - `GetFileList` needs to dispose of the new string list, since the caller does not yet know of its existence. For this reason, initialization of the string list is performed in a `try...except` statement. If an exception occurs, the statement's exception block disposes of the string list, then re-raises the exception.

Nested Exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as these exceptions are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the Tan function below.

```
type
   ETrigError = class(EMathError);
   function Tan(X: Extended): Extended;
   begin
      try
        Result := Sin(X) / Cos(X);
      except
        on EMathError do
        raise ETrigError.Create('Invalid argument to Tan');
      end;
   end;
```

If an EMathError exception occurs during execution of Tan, the exception handler raises an `ETrigError`. Since Tan does not provide a handler for `ETrigError`, the exception propagates beyond the original exception handler, causing the EMathError exception to be destroyed. To the caller, it appears as if the Tan function has raised an `ETrigError` exception.

Try...finally Statements

Sometimes you want to ensure that specific parts of an operation are completed, whether or not the operation is interrupted by an exception. For example, when a routine acquires control of a resource, it is often important that the resource be released, regardless of whether the routine terminates normally. In these situations, you can use a `try...finally` statement.

The following example shows how code that opens and processes a file can ensure that the file is ultimately closed, even if an error occurs during execution.

```
Reset(F);
try
   ... // process file F
finally
   CloseFile(F);
end;
```

The syntax of a `try...finally` statement is

try*statementList1*finally*statementList2*end

where each *statementList* is a sequence of statements delimited by semicolons. The `try...finally` statement executes the statements in *statementList1* (the try clause). If *statementList1* finishes without raising exceptions, *statementList2* (the finally clause) is executed. If an exception is raised during execution of *statementList1*, control is transferred to *statementList2*; once *statementList2* finishes executing, the exception is re-raised. If a call to the `Exit`, `Break`, or `Continue` procedure causes control to leave *statementList1*, *statementList2* is automatically executed. Thus the finally clause is always executed, regardless of how the try clause terminates.

If an exception is raised but not handled in the finally clause, that exception is propagated out of the `try...finally` statement, and any exception already raised in the try clause is lost. The finally clause should therefore handle all locally raised exceptions, so as not to disturb propagation of other exceptions.

Standard Exception Classes and Routines

The SysUtils and
System units declare several standard routines for handling exceptions, including ExceptObject, ExceptAddr, and ShowException. SysUtils, System and other units also include dozens of exception classes, all of which (aside from OutlineError) derive from Exception.

The Exception class has properties called Message and HelpContext that can be used to pass an error description and a context ID for context-sensitive online documentation. It also defines various constructor methods that allow you to specify the description and context ID in different ways.

Nested Type Declarations

Type declarations may be nested within class declarations. Nested types are used throughout the .NET framework, and throughout object-oriented programming in general. They present a way to keep conceptually related types together, and to avoid name collisions. The same syntax for declaring nested types may be used with the Win32 Delphi compiler.

Declaring Nested Types

```
type
  className = class [abstract | sealed] (ancestorType)
      memberList

      type
         nestedTypeDeclaration

      memberList
  end;
```

Where *nestedTypeDeclaration* follows the type declaration syntax defined in Declaring Types.

Nested type declarations are terminated by the first occurance of a non-identifier token, for example, procedure, class, type, and all visibility scope specifiers.

The normal accessibility rules apply to nested types and their containing types. A nested type can access an instance variable (field, property, or method) of its container class, but it must have an object reference to do so. A nested type can access class fields, class properties, and class static methods without an object reference, but the normal Delphi visibility rules apply.

Nested types do not increase the size of the containing class. In other words, creating an instance of the containing class does not also create an instance of a nested type. Nested types are associated with their containing classes only by the context of their declaration.

Declaring and Accessing Nested Classes

The following examples demonstrate how to declare and access fields and methods of a nested class.

```
type
      TOuterClass = class
       strict private
          myField: Integer;

       public
          type
             TInnerClass = class
              public
                myInnerField: Integer;
                procedure innerProc;
             end;

         procedure outerProc;
       end;
```

To implement the `innerProc` method of the inner class, you must qualify its name with the name of the outer class. For example

```
procedure TOuterClass.TInnerClass.innerProc;
begin
   ...
end;
```

To access the members of the nested type, use dotted notation as with regular class member access. For example

```
var
   x: TOuterClass;
   y: TOuterClass.TInnerClass;

begin
   x := TOuterClass.Create;
   x.outerProc;
   ...
   y := TOuterClass.TInnerClass.Create;
   y.innerProc;
```

Nested Constants

Constants can be declared in class types in the same manner as nested type sections. Constant sections are terminated by the same tokens as nested type sections (i.e. reserved words or visibility specifiers). Typed constants are not supported, so you cannot declare nested constants of value types, such as Currency, or TDateTime.

Nested constants can be of any simple type: ordinal, ordinal subranges, enums, strings, and real types.

The following code demonstrates the declaration of nested constants:

```
type
   TMyClass = class
        const
             x = 12;
             y = TMyClass.x + 23;
        procedure Hello;
        private
             const
                 s = 'A string constant';
    end;

begin
   writeln(TMyClass.y);   // Writes the value of y, 35.
end.
```

Standard Routines and I/O

This section describes the standard routines included in the Delphi runtime library.

Standard Routines and I/O

These topics discuss text and file I/O and summarize standard library routines. Many of the procedures and functions listed here are defined in the `System` and *SysInit* units, which are implicitly used with every application. Others are built into the compiler but are treated as if they were in the `System` unit.

Some standard routines are in units such as `SysUtils`, which must be listed in a uses clause to make them available in programs. You cannot, however, list `System` in a uses clause, nor should you modify the `System` unit or try to rebuild it explicitly.

File Input and Output

The table below lists input and output routines.

Input and output procedures and functions

Procedure or function	Description
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.
ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
Readln	Does what Read does and then skips to beginning of next line in the text file.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.
RmDir	Removes an empty subdirectory.
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.
SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.

Writeln	Does the same as Write, and then writes an end-of-line marker to the text file.

A file variable is any variable whose type is a file type. There are three classes of file: typed, text, and untyped. The syntax for declaring file types is given in File types.

Before a file variable can be used, it must be associated with an external file through a call to the AssignFile procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be *opened* to prepare it for input or output. An existing file can be opened via the Reset procedure, and a new file can be created and opened via the Rewrite procedure. Text files opened with Reset are read-only and text files opened with Rewrite and Append are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with Reset or Rewrite.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. The components are numbered starting with zero.

Files are normally accessed sequentially. That is, when a component is read using the standard procedure Read or written using the standard procedure Write, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly through the standard procedure Seek, which moves the current file position to a specified component. The standard functions FilePos and FileSize can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure CloseFile. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be turned on and off using the **{$I+}** and **{$I}** compiler directives. When I/O checking is off, that is, when a procedure or function call is compiled in the **{$I}** state an I/O error doesn't cause an exception to be raised; to check the result of an I/O operation, you must call the standard function IOResult instead.

You must call the IOResult function to clear an error, even if you aren't interested in the error. If you don't clear an error and **{$I+}** is the current state, the next I/O function call will fail with the lingering IOResult error.

Text Files

This section summarizes I/O using file variables of the standard type Text.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a line feed character). The type Text is distinct from the type file of Char.

For text files, there are special forms of Read and Write that let you read and write values that are not of type Char. Such values are automatically translated to and from their character representation. For example, Read(F, I), where I is a type Integer variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in I.

There are two standard text file variables, Input and Output. The standard file variable Input is a read-only file associated with the operating system's standard input (typically, the keyboard). The standard file variable Output is a write-only file associated with the operating system's standard output (typically, the display). Before an application begins executing, Input and Output are automatically opened, as if the following statements were executed:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Note: For Win32 applications, text-oriented I/O is available only in console applications, that is, applications compiled with the Generate console application option checked on the Linker page of the Project Options dialog box or with the -cc command-line compiler option. In a GUI (non-console) application, any attempt to read or write using Input or Output will produce an I/O error.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, Input or Output is assumed by default, depending on whether the procedure or function is input- or output-oriented. For example, Read(X) corresponds to Read(Input, X) and Write (X) corresponds to Write(Output, X).

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using AssignFile, and opened using Reset, Rewrite, or Append. An error occurs if you pass a file that was opened with Reset to an output-oriented procedure or function. An error also occurs if you pass a file that was opened with Rewrite or Append to an input-oriented procedure or function.

Untyped Files

Untyped files are low-level I/O channels used primarily for direct access to disk files regardless of type and structuring. An untyped file is declared with the word file and nothing more. For example,

```
var DataFile: file;
```

For untyped files, the Reset and Rewrite procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file. (No partial records are possible when the record size is 1.)

Except for Read and Write, all typed-file standard procedures and functions are also allowed on untyped files. Instead of Read and Write, two procedures called BlockRead and BlockWrite are used for high-speed data transfers.

Text File Device Drivers

You can define your own text file device drivers for your programs. A text file device driver is a set of four functions that completely implement an interface between Delphi's file system and some device.

The four functions that define each device driver are Open, InOut, Flush, and Close. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where DeviceFunc is the name of the function (that is, Open, InOut, Flush, or Close). The return value of a device-interface function becomes the value returned by IOResult. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized Assign procedure. The Assign procedure must assign the addresses of the four device-interface functions to the four function pointers in the text file variable. In addition, it should store the fmClosed*magic* constant in the Mode field, store the size of the text file buffer in BufSize, store a pointer to the text file buffer in BufPtr, and clear the Name string.

Assuming, for example, that the four device-interface functions are called DevOpen, DevInOut, DevFlush, and DevClose, the Assign procedure might look like this:

```
procedure AssignDev(var F: Text);
  begin
   with TTextRec(F) do
   begin
    Mode := fmClosed;
```

```
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
   end;
  end;
```

The device-interface functions can use the UserData field in the file record to store private information. This field isn't modified by the product file system at any time.

The Open function

The Open function is called by the Reset, Rewrite, and Append standard procedures to open a text file associated with a device. On entry, the Mode field contains fmInput, fmOutput, or fmInOut to indicate whether the Open function was called from Reset, Rewrite, or Append.

The Open function prepares the file for input or output, according to the Mode value. If Mode specified fmInOut (indicating that Open was called from Append), it must be changed to fmOutput before Open returns.

Open is always called before any of the other device-interface functions. For that reason, AssignDev only initializes the OpenFunc field, leaving initialization of the remaining vectors up to Open. Based on Mode, Open can then install pointers to either input- or output-oriented functions. This saves the InOut, Flush functions and the CloseFile procedure from determining the current mode.

The InOut function

The InOut function is called by the Read, Readln, Write, Writeln, Eof, Eoln, SeekEof, SeekEoln, and CloseFile standard routines whenever input or output from the device is required.

When Mode is fmInput, the InOut function reads up to BufSize characters into BufPtr^, and returns the number of characters read in BufEnd. In addition, it stores zero in BufPos. If the InOut function returns zero in BufEnd as a result of an input request, Eof becomes **True** for the file.

When Mode is fmOutput, the InOut function writes BufPos characters from BufPtr^, and returns zero in BufPos.

The Flush function

The Flush function is called at the end of each Read, Readln, Write, and Writeln. It can optionally flush the text file buffer.

If Mode is fmInput, the Flush function can store zero in BufPos and BufEnd to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If Mode is fmOutput, the Flush function can write the contents of the buffer exactly like the InOut function, which ensures that text written to the device appears on the device immediately. If Flush does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.

The Close function

The Close function is called by the CloseFile standard procedure to close a text file associated with a device. (The Reset, Rewrite, and Append procedures also call Close if the file they are opening is already open.) If Mode is fmOutput, then before calling Close, the file system calls the InOut function to ensure that all characters have been written to the device.

Handling null-Terminated Strings

The Delphi language's extended syntax allows the Read, Readln, Str, and Val standard procedures to be applied to zero-based character arrays, and allows the Write, Writeln, Val, AssignFile, and Rename standard procedures to be applied to both zero-based character arrays and character pointers.

Null-Terminated String Functions

The following functions are provided for handling null-terminated strings.

Null-terminated string functions

Function	Description
StrAlloc	Allocates a character buffer of a given size on the heap.
StrBufSize	Returns the size of a character buffer allocated using StrAlloc or StrNew.
StrCat	Concatenates two strings.
StrComp	Compares two strings.
StrCopy	Copies a string.
StrDispose	Disposes a character buffer allocated using StrAlloc or StrNew.
StrECopy	Copies a string and returns a pointer to the end of the string.
StrEnd	Returns a pointer to the end of a string.
StrFmt	Formats one or more values into a string.
StrIComp	Compares two strings without case sensitivity.
StrLCat	Concatenates two strings with a given maximum length of the resulting string.
StrLComp	Compares two strings for a given maximum length.
StrLCopy	Copies a string up to a given maximum length.
StrLen	Returns the length of a string.
StrLFmt	Formats one or more values into a string with a given maximum length.
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrLower	Converts a string to lowercase.
StrMove	Moves a block of characters from one string to another.
StrNew	Allocates a string on the heap.
StrPCopy	Copies a Pascal string to a null-terminated string.
StrPLCopy	Copies a Pascal string to a null-terminated string with a given maximum length.
StrPos	Returns a pointer to the first occurrence of a given substring within a string.
StrRScan	Returns a pointer to the last occurrence of a given character within a string.
StrScan	Returns a pointer to the first occurrence of a given character within a string.
StrUpper	Converts a string to uppercase.

Standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. Names of multibyte functions start with Ansi-. For example, the multibyte version of StrPos is AnsiStrPos. Multibyte character support is operating-system dependent and based on the current locale.

Wide-Character Strings

The `System` unit provides three functions, WideCharToString, WideCharLenToString, and StringToWideChar, that can be used to convert null-terminated wide character strings to single- or double-byte long strings.

Assignment will also convert between strings. For instance, the following are both valid:

```
MyAnsiString := MyWideString;
     MyWideString := MyAnsiString;
```

Other Standard Routines

The table below lists frequently used procedures and functions found in Borland product libraries. This is not an exhaustive inventory of standard routines.

Other standard routines

Procedure or function	Description
Addr	Returns a pointer to a specified object.
AllocMem	Allocates a memory block and initializes each byte to zero.
ArcTan	Calculates the arctangent of the given number.
Assert	Raises an exception if the passed expression does not evaluate to true.
Assigned	Tests for a **nil** (unassigned) pointer or procedural variable.
Beep	Generates a standard beep.
Break	Causes control to exit a for, while, or repeat statement.
ByteToCharIndex	Returns the position of the character containing a specified byte in a string.
Chr	Returns the character for a specified integer value.
Close	Closes a file.
CompareMem	Performs a binary comparison of two memory images.
CompareStr	Compares strings case sensitively.
CompareText	Compares strings by ordinal value and is not case sensitive.
Continue	Returns control to the next iteration of for, while, or repeat statements.
Copy	Returns a substring of a string or a segment of a dynamic array.
Cos	Calculates the cosine of an angle.
CurrToStr	Converts a currency variable to a string.
Date	Returns the current date.
DateTimeToStr	Converts a variable of type TDateTime to a string.
DateToStr	Converts a variable of type TDateTime to a string.
Dec	Decrements an ordinal variable or a typed pointer variable.
Dispose	Releases dynamically allocated variable memory.
ExceptAddr	Returns the address at which the current exception was raised.
Exit	Exits from the current procedure.
Exp	Calculates the exponential of X.
FillChar	Fills contiguous bytes with a specified value.

Finalize	Uninitializes a dynamically allocated variable.
FloatToStr	Converts a floating point value to a string.
FloatToStrF	Converts a floating point value to a string, using specified format.
FmtLoadStr	Returns formatted output using a resourced format string.
FmtStr	Assembles a formatted string from a series of arrays.
Format	Assembles a string from a format string and a series of arrays.
FormatDateTime	Formats a date-and-time value.
FormatFloat	Formats a floating point value.
FreeMem	Releases allocated memory.
GetMem	Allocates dynamic memory and a pointer to the address of the block.
Halt	Initiates abnormal termination of a program.
Hi	Returns the high-order byte of an expression as an unsigned value.
High	Returns the highest value in the range of a type, array, or string.
Inc	Increments an ordinal variable or a typed pointer variable.
Initialize	Initializes a dynamically allocated variable.
Insert	Inserts a substring at a specified point in a string.
Int	Returns the integer part of a real number.
IntToStr	Converts an integer to a string.
Length	Returns the length of a string or array.
Lo	Returns the low-order byte of an expression as an unsigned value.
Low	Returns the lowest value in the range of a type, array, or string.
LowerCase	Converts an ASCII string to lowercase.
MaxIntValue	Returns the largest signed value in an integer array.
MaxValue	Returns the largest signed value in an array.
MinIntValue	Returns the smallest signed value in an integer array.
MinValue	Returns smallest signed value in an array.
New	Creates a dynamic allocated variable memory and references it with a specified pointer.
Now	Returns the current date and time.
Ord	Returns the ordinal integer value of an ordinal-type expression.
Pos	Returns the index of the first single-byte character of a specified substring in a string.
Pred	Returns the predecessor of an ordinal value.
Ptr	Converts a value to a pointer.
Random	Generates random numbers within a specified range.
ReallocMem	Reallocates a dynamically allocatable memory.
Round	Returns the value of a real rounded to the nearest whole number.
SetLength	Sets the dynamic length of a string variable or array.
SetString	Sets the contents and length of the given string.
ShowException	Displays an exception message with its address.

ShowMessage	Displays a message box with an unformatted string and an OK button.
ShowMessageFmt	Displays a message box with a formatted string and an OK button.
Sin	Returns the sine of an angle in radians.
SizeOf	Returns the number of bytes occupied by a variable or type.
Sqr	Returns the square of a number.
Sqrt	Returns the square root of a number.
Str	Converts an integer or real number into a string.
StrToCurr	Converts a string to a currency value.
StrToDate	Converts a string to a date format (TDateTime).
StrToDateTime	Converts a string to a TDateTime.
StrToFloat	Converts a string to a floating-point value.
StrToInt	Converts a string to an integer.
StrToTime	Converts a string to a time format (TDateTime).
StrUpper	Returns an ASCII string in upper case.
Succ	Returns the successor of an ordinal value.
Sum	Returns the sum of the elements from an array.
Time	Returns the current time.
TimeToStr	Converts a variable of type TDateTime to a string.
Trunc	Truncates a real number to an integer.
UniqueString	Ensures that a string has only one reference. (The string may be copied to produce a single reference.)
UpCase	Converts a character to uppercase.
UpperCase	Returns a string in uppercase.
VarArrayCreate	Creates a variant array.
VarArrayDimCount	Returns number of dimensions of a variant array.
VarArrayHighBound	Returns high bound for a dimension in a variant array.
VarArrayLock	Locks a variant array and returns a pointer to the data.
VarArrayLowBound	Returns the low bound of a dimension in a variant array.
VarArrayOf	Creates and fills a one-dimensional variant array.
VarArrayRedim	Resizes a variant array.
VarArrayRef	Returns a reference to the passed variant array.
VarArrayUnlock	Unlocks a variant array.
VarAsType	Converts a variant to specified type.
VarCast	Converts a variant to a specified type, storing the result in a variable.
VarClear	Clears a variant.
VarCopy	Copies a variant.
VarToStr	Converts variant to string.
VarType	Returns type code of specified variant.

Libraries and Packages

This section describes how to create static and dynamically loadable libraries in Delphi.

Libraries and Packages

A dynamically loadable library is a dynamic-link library (DLL) on Win32, and an assembly (also a DLL) on the .NET platform. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it.

Delphi programs can call DLLs and assemblies written in other languages, and applications written in other languages can call DLLs or assemblies written in Delphi.

Calling Dynamically Loadable Libraries

You can call operating system routines directly, but they are not linked to your application until runtime. This means that the library need not be present when you compile your program. It also means that there is no compile-time validation of attempts to import a routine.

Before you can call routines defined in DLL or assembly, you must import them. This can be done in two ways: by declaring an external procedure or function, or by direct calls to the operating system. Whichever method you use, the routines are not linked to your application until runtime.

The Delphi language does not support importing of variables from DLLs or assemblies.

Static Loading

The simplest way to import a procedure or function is to declare it using the external directive. For example,

```
procedure DoSomething; external 'MYLIB.DLL';
```

If you include this declaration in a program, MYLIB.DLL is loaded once, when the program starts. Throughout execution of the program, the identifier `DoSomething` always refers to the same entry point in the same shared library.

Declarations of imported routines can be placed directly in the program or unit where they are called. To simplify maintenance, however, you can collect external declarations into a separate "import unit" that also contains any constants and types required for interfacing with the library. Other modules that use the import unit can call any routines declared in it.

Dynamic Loading

You can access routines in a library through direct calls to Win32 APIs, including `LoadLibrary`, `FreeLibrary`, and `GetProcAddress`. These functions are declared in Windows.pas. on Linux, they are implemented for compatibility in SysUtils.pas; the actual Linux OS routines are *dlopen*, *dlclose*, and *dlsym* (all declared in libc; see the man pages for more information). In this case, use procedural-type variables to reference the imported routines.

For example,

```
uses Windows, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
```

```
TGetTime = procedure(var Time: TTimeRec);
THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
              .
              .
              .
begin
  Handle := LoadLibrary('libraryname');
  if Handle <> 0 then
  begin
   @GetTime := GetProcAddress(Handle, 'GetTime');
   if @GetTime <> nil then
      begin
       GetTime(Time);
           with Time do
               WriteLn('The time is ', Hour, ':', Minute, ':', Second);
   end;
   FreeLibrary(Handle);
  end;
end;
```

When you import routines this way, the library is not loaded until the code containing the call to `LoadLibrary` executes. The library is later unloaded by the call to `FreeLibrary`. This allows you to conserve memory and to run your program even when some of the libraries it uses are not present.

Writing Dynamically Loaded Libraries

The following topics describe elements of writing dynamically loadable libraries, including

- The exports clause.
- Library initialization code.
- Global variables.
- Libraries and system variables.

Using Export Clause in Libraries

The main source for a dynamically loadable library is identical to that of a program, except that it begins with the reserved word library (instead of program).

Only routines that a library explicitly exports are available for importing by other libraries or programs. The following example shows a library with two exported functions, `Min` and `Max`.

```
library MinMax;
    function Min(X, Y: Integer): Integer; stdcall;
 begin
   if X < Y then Min := X else Min := Y;
 end;
 function Max(X, Y: Integer): Integer; stdcall;
 begin
   if X > Y then Max := X else Max := Y;
 end;
   exports
      Min,
   Max;
 begin
 end.
```

If you want your library to be available to applications written in other languages, it's safest to specify stdcall in the declarations of exported functions. Other languages may not support Delphi's default register calling convention.

Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a uses clause, an exports clause, and the initialization code. For example,

```
library Editors;
  uses EdInit, EdInOut, EdFormat, EdPrint;
  exports
    InitEditors,
    DoneEditors name Done,
    InsertText name Insert,
    DeleteSelection name Delete,
    FormatSelection,
    PrintSelection name Print,
    .
                .
                .
    SetErrorHandler;
  begin
    InitLibrary;
  end.
```

You can put exports clauses in the interface or implementation section of a unit. Any library that includes such a unit in its uses clause automatically exports the routines listed the unit's exports clauses without the need for an exports clause of its own.

The directive local, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming.

On Linux, the local directive provides a slight performance optimization for routines that are compiled into a library but are not exported. This directive can be specified for stand-alone procedures and functions, but not for methods. A routine declared with localfor example,

```
function Contraband(I: Integer): Integer; local;
```

does not refresh the EBX register and hence

- cannot be exported from a library.
- cannot be declared in the interface section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the caller sets up EBX.

A routine is exported when it is listed in an exports clause, which has the form

```
exports entry1, ..., entryn;
```

where each entry consists of the name of a procedure, function, or variable (which must be declared prior to the exports clause), followed by a parameter list (only if exporting a routine that is overloaded), and an optional name specifier. You can qualify the procedure or function name with the name of a unit.

(Entries can also include the directive resident, which is maintained for backward compatibility and is ignored by the compiler.)

On the Win32 platform, an index specifier consists of the directive index followed by a numeric constant between 1 and 2,147,483,647. (For more efficient programs, use low index values.) If an entry has no index specifier, the routine is automatically assigned a number in the export table.

Note: Use of index specifiers, which are supported for backward compatibility only, is discouraged and may cause problems for other development tools.

A name specifier consists of the directive name followed by a string constant. If an entry has no name specifier, the routine is exported under its original declared name, with the same spelling and case. Use a name clause when you want to export a routine under a different name. For example,

```
exports
DoSomethingABC name 'DoSomething';
```

When you export an overloaded function or procedure from a dynamically loadable library, you must specify its parameter list in the exports clause. For example,

```
exports
Divide(X, Y: Integer) name 'Divide_Ints',
Divide(X, Y: Real) name 'Divide_Reals';
```

On Win32, do not include index specifiers in entries for overloaded routines.

An exports clause can appear anywhere and any number of times in the declaration part of a program or library, or in the interface or implementation section of a unit. Programs seldom contain an exports clause.

Library Initialization Code

The statements in a library's block constitute the library's initialization code. These statements are executed once every time the library is loaded. They typically perform tasks like registering window classes and initializing variables. Library initialization code can also install an entry point procedure using the `DllProc` variable. The `DllProc` variable is similar to an exit procedure, which is described in Exit procedures; the entry point procedure executes when the library is loaded or unloaded.

Library initialization code can signal an error by setting the `ExitCode` variable to a nonzero value. ExitCode is declared in the `System` unit and defaults to zero, indicating successful initialization. If a library's initialization code sets `ExitCode` to another value, the library is unloaded and the calling application is notified of the failure. Similarly, if an unhandled exception occurs during execution of the initialization code, the calling application is notified of a failure to load the library.

Here is an example of a library with initialization code and an entry point procedure.

```
library Test;
var
 SaveDllProc: Pointer;
 procedure LibExit(Reason: Integer);
begin
 if Reason = DLL_PROCESS_DETACH then
  begin
    .
                .   // library exit code
                .
  end;
   SaveDllProc(Reason);        // call saved entry point procedure
  end;
 begin
    .
                .              // library initialization code
                .
  SaveDllProc := DllProc;         // save exit procedure chain
  DllProc := @LibExit;         // install LibExit exit procedure
 end.
```

DllProc is called when the library is first loaded into memory, when a thread starts or stops, or when the library is unloaded. The initialization parts of all units used by a library are executed before the library's initialization code, and the finalization parts of those units are executed after the library's entry point procedure.

Global Variables in a Library

Global variables declared in a shared library cannot be imported by a Delphi application.

A library can be used by several applications at once, but each application has a copy of the library in its own process space with its own set of global variables. For multiple libraries - or multiple instances of a library - to share memory, they must use memory-mapped files. Refer to the your system documentation for further information.

Libraries and System Variables

Several variables declared in the System unit are of special interest to those programming libraries. Use IsLibrary to determine whether code is executing in an application or in a library; IsLibrary is always False in an application and True in a library. During a library's lifetime, HInstance contains its instance handle. CmdLine is always nil in a library.

The DLLProc variable allows a library to monitor calls that the operating system makes to the library entry point. This feature is normally used only by libraries that support multithreading. DLLProc is available on both Windows and Linux but its use differs on each. On Win32, DLLProc is used in multithreading applications.; on Linux, it is used to determine when your library is being unloaded. You should use finalization sections, rather than exit procedures, for all exit behavior.

To monitor operating-system calls, create a callback procedure that takes a single integer parameterfor example,

```
procedure DLLHandler(Reason: Integer);
```

and assign the address of the procedure to the DLLProc variable. When the procedure is called, it passes to it one of the following values.

DLL_PROCESS_DETACH	Indicates that the library is detaching from the address space of the calling process as a result of a clean exit or a call to FreeLibrary.
DLL_PROCESS_ATTACH	Indicates that the library is attaching to the address space of the calling process as the result of a call to LoadLibrary.
DLL_THREAD_ATTACH	Indicates that the current process is creating a new thread.
DLL_THREAD_DETACH	Indicates that a thread is exiting cleanly.

In the body of the procedure, you can specify actions to take depending on which parameter is passed to the procedure.

Exceptions and Runtime Errors in Libraries

When an exception is raised but not handled in a dynamically loadable library, it propagates out of the library to the caller. If the calling application or library is itself written in Delphi, the exception can be handled through a normal try...except statement.

On Win32, if the calling application or library is written in another language, the exception can be handled as an operating-system exception with the exception code $0EEDFADE. The first entry in the ExceptionInformation array of the operating-system exception record contains the exception address, and the second entry contains a reference to the Delphi exception object.

Generally, you should not let exceptions escape from your library. Delphi exceptions map to the OS exception model (including the .NET exception model)..

If a library does not use the `SysUtils` unit, exception support is disabled. In this case, when a runtime error occurs in the library, the calling application terminates. Because the library has no way of knowing whether it was called from a Delphi program, it cannot invoke the application's exit procedures; the application is simply aborted and removed from memory.

Shared-Memory Manager (Win32 Only)

On Win32, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all use the ShareMem unit. The same is true if one application or DLL allocates memory with `New` or `GetMem` which is deallocated by a call to Dispose or FreeMem in another module. ShareMem should always be the first unit listed in any program or library uses clause where it occurs.

ShareMem is the interface unit for the BORLANDMM.DLL memory manager, which allows modules to share dynamically allocated memory. BORLANDMM.DLL must be deployed with applications and DLLs that use ShareMem. When an application or DLL uses ShareMem, its memory manager is replaced by the memory manager in BORLANDMM.DLL.

Packages

The following topics describe packages and various issues involved in creating and compiling them.

- Package declarations and source files
- Naming packages
- The requires clause
- Avoiding circular package references
- Duplicate package references
- The contains clause
- Avoiding redundant source code uses
- Compiling packages
- Generated files
- Package-specific compiler directives
- Package-specific command-line compiler switches

Understanding Packages

A package is a specially compiled library used by applications, the IDE, or both. Packages allow you to rearrange where code resides without affecting the source code. This is sometimes referred to as *application partitioning*.

Runtime packages provide functionality when a user runs an application. Design-time packages are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by referencing runtime packages in their requires clauses.

On Win32, package files end with the .bpl (Borland package library) extension. On the .NET platform, packages are .NET assemblies, and end with an extension of .dll

Ordinarily, packages are loaded statically when an applications starts. But you can use the `LoadPackage` and `UnloadPackage` routines (in the `SysUtils` unit) to load packages dynamically.

Note: When an application utilizes packages, the name of each packaged unit still must appear in the uses clause of any source file that references it.

Package Declarations and Source Files

Each package is declared in a separate source file, which should be saved with the .dpk extension to avoid confusion with other files containing Delphi code. A package source file does not contain type, data, procedure, or function declarations. Instead, it contains:

- a name for the package.
- a list of other packages required by the new package. These are packages to which the new package is linked.
- a list of unit files contained by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which provide the functionality of the compiled package.

A package declaration has the form

package*packageName;*

requiresClause;

containsClause;

end.

where *packageName* is any valid identifier. The *requiresClause* and *containsClause* are both optional. For example, the following code declares the `DATAX` package.

```
package DATAX;
  requires
  rtl,
  contains Db, DBLocal, DBXpress, ... ;
end.
```

The requires clause lists other, external packages used by the package being declared. It consists of the directive requires, followed by a comma-delimited list of package names, followed by a semicolon. If a package does not reference other packages, it does not need a requires clause.

The contains clause identifies the unit files to be compiled and bound into the package. It consists of the directive contains, followed by a comma-delimited list of unit names, followed by a semicolon. Any unit name may be followed by the reserved word in and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. For example,

```
contains MyUnit in 'C:\MyProject\MyUnit.pas';
```

Note: Thread-local variables (declared with threadvar) in a packaged unit cannot be accessed from clients that use the package.

Naming packages

A compiled package involves several generated files. For example, the source file for the package called DATAX is DATAX.DPK, from which the compiler generates an executable and a binary image called

DATAX.BPL (Win32) or DATAX.DLL (.NET), and DATAX.DCP (Win32) or DATAX.DCPIL (.NET)

DATAX is used to refer to the package in the requires clauses of other packages, or when using the package in an application. Package names must be unique within a project.

The requires clause

The requires clause lists other, external packages that are used by the current package. It functions like the uses clause in a unit file. An external package listed in the requires clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in a package make references to other packaged units, the other packages should be included in the first package's requires clause. If the other packages are omitted from the requires clause, the compiler loads the referenced units from their .dcu or .dcuil files.

Avoiding circular package references

Packages cannot contain circular references in their requires clauses. This means that

- A package cannot reference itself in its own requires clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Duplicate package references

The compiler ignores duplicate references in a package's requires clause. For programming clarity and readability, however, duplicate references should be removed.

The contains clause

The contains clause identifies the unit files to be bound into the package. Do not include file-name extensions in the contains clause.

Avoiding redundant source code uses

A package cannot be listed in the contains clause of another package or the uses clause of a unit.

All units included directly in a package's contains clause, or indirectly in the uses clauses of those units, are bound into the package at compile time. The units contained (directly or indirectly) in a package cannot be contained in any other packages referenced in requires clause of that package.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application.

Compiling Packages

Packages are ordinarily compiled from the IDE using .dpk files generated by the **Project Manager**. You can also compile .dpk files directly from the command line. When you build a project that contains a package, the package is implicitly recompiled, if necessary.

Generated Files

The following table lists the files produced by the successful compilation of a package.

Compiled package files

File extension	Contents
DCP (Win32) or DCPIL (.NET)	A binary image containing a package header and the concatenation of all .dcu (Win32) or .dcuil (.NET) files in the package. A single .dcp or .dcpil file is created for each package. The base name for the file is the base name of the .dpk source file.
BPL (Win32) or DLL (.NET)	The runtime package. This file is a DLL on Win32 with special Borland-specific features. The base name for the package is the base name of the dpk source file.

Package-Specific Compiler Directives

The following table lists package-specific compiler directives that can be inserted into source code.

Package-specific compiler directives

Directive	Purpose
`{$IMPLICITBUILD OFF}`	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
`{$G-}` or `{$IMPORTEDDATA OFF}`	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
`{$WEAKPACKAGEUNIT ON}`	Packages unit weakly.
`{$DENYPACKAGEUNIT ON}`	Prevents unit from being placed in a package.

{$DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .dpk file.)
{$RUNONLY ON}	Compiles the package as runtime only. (Put in .dpk file.)

Including {$DENYPACKAGEUNIT ON} in source code prevents the unit file from being packaged. Including {$G-} or {$IMPORTEDDATA OFF} may prevent a package from being used in the same application with other packages.

Other compiler directives may be included, if appropriate, in package source code.

Package-Specific Command-Line Compiler Switches

The following package-specific switches are available for the command-line compiler.

Package-specific command-line compiler switches

Switch	Purpose
-$G-	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
LE path	Specifies the directory where the compiled package file will be placed.
LN path	Specifies the directory where the package dcp or dcpil file will be placed.
LUpackageName [;packageName2;...]	Specifies additional runtime packages to use in an application. Used when compiling a project.
Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Using the -$G- switch may prevent a package from being used in the same application with other packages.

Other command-line options may be used, if appropriate, when compiling packages.

Note: When using the -LU switch on the .NET platform, you can refer to the package with or without the .dll extension. If you omit the .dll extension, the compiler will look for the package on the unit search path, and on the package search path. However, if the package specification contains a drive letter or the path separator character, then the compiler will assume the package name is the full file name (including the .dll extension). In the latter case, if you specify a full or relative path, but omit the .dll extension, the compiler will not be able to locate the package.

Object Interfaces

This section describes the use of interfaces in Delphi.

Object Interfaces

An object interface, or simply interface, defines methods that can be implemented by a class. Interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface's methods. A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable.

Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using distributed object models (such as CORBA and SOAP). Using a distributed object model, custom objects that support interfaces can interact with objects written in C++, Java, and other languages.

Interface Types

Interfaces, like classes, can be declared only in the outermost scope of a program or unit, not in a procedure or function declaration. An interface type declaration has the form

```
type interfaceName = interface (ancestorInterface)
        ['{GUID}']
        memberList
end;
```

where *(ancestorInterface)* and *['{GUID}']* are optional. In most respects, interface declarations resemble class declarations, but the following restrictions apply.

- The *memberList* can include only methods and properties. Fields are not allowed in interfaces.
- Since an interface has no fields, property read and write specifiers must be methods.
- All members of an interface are public. Visibility specifiers and storage specifiers are not allowed. (But an array property can be declared as default.)
- Interfaces have no constructors or destructors. They cannot be instantiated, except through classes that implement their methods.
- Methods cannot be declared as virtual, dynamic, abstract, or override. Since interfaces do not implement their own methods, these designations have no meaning.

Here is an example of an interface declaration:

```
type
IMalloc = interface(IInterface)
        ['{00000002-0000-0000-C000-000000000046}']
        function Alloc(Size: Integer): Pointer; stdcall;
        function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
        procedure Free(P: Pointer); stdcall;
        function GetSize(P: Pointer): Integer; stdcall;
        function DidAlloc(P: Pointer): Integer; stdcall;
        procedure HeapMinimize; stdcall;
end;
```

In some interface declarations, the interface reserved word is replaced by dispinterface. This construction (along with the dispid, read only, and write only directives) is platform-specific and is not used in Linux programming.

IInterface and Inheritance

An interface, like a class, inherits all of its ancestors' methods. But interfaces, unlike classes, do not implement methods. What an interface inherits is the obligation to implement methods an obligation that is passed onto any class supporting the interface.

The declaration of an interface can specify an ancestor interface. If no ancestor is specified, the interface is a direct descendant of `IInterface`, which is defined in the `System` unit and is the ultimate ancestor of all other interfaces. On Win32, `IInterface` declares three methods: `QueryInterface`, `_AddRef`, and `_Release`. These methods are not present on the .NET platform, and you do not need to implement them.

Note: `IInterface` is equivalent to `IUnknown`. You should generally use `IInterface` for platform independent applications and reserve the use of `IUnknown` for specific programs that include Win32 dependencies.

`QueryInterface` provides the means to obtain a reference to the different interfaces that an object supports. `_AddRef` and `_Release` provide lifetime memory management for interface references. The easiest way to implement these methods is to derive the implementing class from the `System` unit's TInterfacedObject. It is also possible to dispense with any of these methods by implementing it as an empty function; COM objects, however, must be managed through `_AddRef` and `_Release`.

Interface Identification

An interface declaration can specify a globally unique identifier (GUID), represented by a string literal enclosed in brackets immediately preceding the member list. The GUID part of the declaration must have the form

['{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}']

where each x is a hexadecimal digit (0 through 9 or A through F). The Type Library editor automatically generates GUIDs for new interfaces. You can also generate GUIDs by pressing Ctrl+Shift+G in the code editor.

A GUID is a 16-byte binary value that uniquely identifies an interface. If an interface has a GUID, you can use interface querying to get references to its implementations.

The TGUID and PGUID types, declared in the `System` unit, are used to manipulate GUIDs.

```
type
    PGUID = ^TGUID;
    TGUID = packed record
        D1: Longword;
        D2: Word;
        D3: Word;
        D4: array[0..7] of Byte;
end;
```

On the .NET platform, you can tag an interface as described above (i.e. following the interface declaration). However, if you use the traditional Delphi syntax, the first square bracket construct following the interface declaration is taken as a GUID specifier - not as a .NET attribute. (Note that .NET attributes always apply to the *next* symbol, not the previous one.) You can also associate a GUID with an interface using the .NET Guid custom attribute. In this case you would use the .NET style syntax, placing the attribute immediately before the interface declaration.

GUIDs are not required for interfaces in the .NET framework. They are only used for COM interoperability.

When you declare a typed constant of type TGUID, you can use a string literal to specify its value. For example,

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

In procedure and function calls, either a GUID or an interface identifier can serve as a value or constant parameter of type TGUID. For example, given the declaration

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports can be called in either of two ways

```
if Supports(Allocator, IMalloc) then ...
```

or

```
if Supports(Allocator, IID_IMalloc) then ...
```

Calling Conventions for Interfaces

The default calling convention for interface methods is register, but interfaces shared among modules (especially if they are written in different languages) should declare all methods with stdcall. Use safecall to implement CORBA interfaces. On Win32, you can use safecall to implement methods of dual interfaces.

Interface Properties

Properties declared in an interface are accessible only through expressions of the interface type; they cannot be accessed through class-type variables. Moreover, interface properties are visible only within programs where the interface is compiled.

In an interface, property read and write specifiers must be methods, since fields are not available.

Forward Declarations

An interface declaration that ends with the reserved word interface and a semicolon, without specifying an ancestor, GUID, or member list, is a forward declaration. A forward declaration must be resolved by a defining declaration of the same interface within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent interfaces. For example,

```
type
     IControl = interface;
     IWindow = interface
     ['{00000115-0000-0000-C000-000000000044}']
     function GetControl(Index: Integer): IControl;
                    .
                    .
                    .
     end;
     IControl = interface
     ['{00000115-0000-0000-C000-000000000049}']
     function GetWindow: IWindow;
     .
               .
               .
     end;
```

Mutually derived interfaces are not allowed. For example, it is not legal to derive `IWindow` from `IControl` and also derive `IControl` from `IWindow`.

Implementing Interfaces

Once an interface has been declared, it must be implemented in a class before it can be used. The interfaces implemented by a class are specified in the class's declaration, after the name of the class's ancestor.

Class Declarations

Such declarations have the form

```
type className = class (ancestorClass, interface1, ..., interfacen)
    memberList
end;
```

For example,

```
type
    TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    .
    .
    .
end;
```

declares a class called `TMemoryManager` that implements the `IMalloc` and `IErrorInfo` interfaces. When a class implements an interface, it must implement (or inherit an implementation of) each method declared in the interface.

Here is the (Win32) declaration of TInterfacedObject in the `System` unit. On the .NET platform, TInterfacedObject is an alias for TObject.

```
     type
        TInterfacedObject = class(TObject, IInterface)
        protected
        FRefCount: Integer;
        function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
        function _AddRef: Integer; stdcall;
        function _Release: Integer; stdcall;
        public
        procedure AfterConstruction; override;
        procedure BeforeDestruction; override;
        class function NewInstance: TObject; override;
        property RefCount: Integer read FRefCount;
    end;
```

TInterfacedObject implements the IInterface interface. Hence TInterfacedObject declares and implements each of the three IInterface methods.

Classes that implement interfaces can also be used as base classes. (The first example above declares `TMemoryManager` as a direct descendent of TInterfacedObject.) On the Win32 platform, every interface inherits from IInterface, and a class that implements interfaces must implement the `QueryInterface`, `_AddRef`, and `_Release` methods. The `System` unit's TInterfacedObject implements these methods and is thus a convenient base from which to derive other classes that implement interfaces. On the .NET platform, IInterface does not declare these methods, and you do not need to implement them.

When an interface is implemented, each of its methods is mapped onto a method in the implementing class that has the same result type, the same calling convention, the same number of parameters, and identically typed parameters

in each position. By default, each interface method is mapped to a method of the same name in the implementing class.

Method Resolution Clause

You can override the default name-based mappings by including method resolution clauses in a class declaration. When a class implements two or more interfaces that have identically named methods, use method resolution clauses to resolve the naming conflicts.

A method resolution clause has the form

```
procedure interface.interfaceMethod = implementingMethod;
```

or

```
function interface.interfaceMethod = implementingMethod;
```

where *implementingMethod* is a method declared in the class or one of its ancestors. The *implementingMethod* can be a method declared later in the class declaration, but cannot be a private method of an ancestor class declared in another module.

For example, the class declaration

```
type
    TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    .
    .
    .
end;
```

maps `IMalloc`'s `Alloc` and `Free` methods onto `TMemoryManager`'s `Allocate` and `Deallocate` methods.

A method resolution clause cannot alter a mapping introduced by an ancestor class.

Changing Inherited Implementations

Descendant classes can change the way a specific interface method is implemented by overriding the implementing method. This requires that the implementing method be virtual or dynamic.

A class can also reimplement an entire interface that it inherits from an ancestor class. This involves relisting the interface in the descendant class' declaration. For example,

```
type
    IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    .
    .
    .
end;
    TWindow = class(TInterfacedObject, IWindow)// TWindow implements IWindow
```

```
    procedure Draw;
    .
    .
    .
end;
TFrameWindow = class(TWindow, IWindow)// TFrameWindow reimplements IWindow
    procedure Draw;
    .
    .
    .
end;
```

Reimplementing an interface hides the inherited implementation of the same interface. Hence method resolution clauses in an ancestor class have no effect on the reimplemented interface.

Implementing Interfaces by Delegation

The **implements** directive allows you to delegate implementation of an interface to a property in the implementing class. For example,

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

declares a property called `MyInterface` that implements the interface `IMyInterface`.

The implements directive must be the last specifier in the property declaration and can list more than one interface, separated by commas. The delegate property

- must be of a class or interface type.
- cannot be an array property or have an index specifier.
- must have a read specifier. If the property uses a read method, that method must use the default register calling convention and cannot be dynamic (though it can be virtual) or specify the message directive.

The class you use to implement the delegated interface should derive from `TAggregationObject`.

Delegating to an Interface-Type Property

If the delegate property is of an interface type, that interface, or an interface from which it derives, must occur in the ancestor list of the class where the property is declared. The delegate property must return an object whose class completely implements the interface specified by the implements directive, and which does so without method resolution clauses. For example,

```
type
 IMyInterface = interface
 procedure P1;
    procedure P2;
end;
TMyClass = class(TObject, IMyInterface)
 FMyInterface: IMyInterface;
 property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
end;
var
 MyClass: TMyClass;
 MyInterface: IMyInterface;
begin
 MyClass := TMyClass.Create;
```

```
 MyClass.FMyInterface := ...// some object whose class implements IMyInterface
 MyInterface := MyClass;
 MyInterface.P1;
end;
```

Delegating to a Class-Type Property

If the delegate property is of a class type, that class and its ancestors are searched for methods implementing the specified interface before the enclosing class and its ancestors are searched. Thus it is possible to implement some methods in the class specified by the property, and others in the class where the property is declared. Method resolution clauses can be used in the usual way to resolve ambiguities or specify a particular method. An interface cannot be implemented by more than one class-type property. For example,

```
type
 IMyInterface = interface
 procedure P1;
 procedure P2;
end;
TMyImplClass = class
 procedure P1;
 procedure P2;
end;
 TMyClass = class(TInterfacedObject, IMyInterface)
 FMyImplClass: TMyImplClass;
 property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
 procedure IMyInterface.P1 = MyP1;
 procedure MyP1;
end;
 procedure TMyImplClass.P1;
 .
    .
    .
 procedure TMyImplClass.P2;
 .
    .
 .
 procedure TMyClass.MyP1;
    .
    .
    .
var
 MyClass: TMyClass;
 MyInterface: IMyInterface;
begin
 MyClass := TMyClass.Create;
 MyClass.FMyImplClass := TMyImplClass.Create;
 MyInterface := MyClass;
 MyInterface.P1;    // calls TMyClass.MyP1;
 MyInterface.P2;    // calls TImplClass.P2;
end;
```

Interface References

If you declare a variable of an interface type, the variable can reference instances of any class that implements the interface. These topics describe Interface references and related topics.

Implementing Interface References

Interface reference variables allow you to call interface methods without knowing at compile time where the interface is implemented. But they are subject to the following:

- An interface-type expression gives you access only to methods and properties declared in the interface, not to other members of the implementing class.
- An interface-type expression cannot reference an object whose class implements a descendant interface, unless the class (or one that it inherits from) explicitly implements the ancestor interface as well.

For example,

```
type
    IAncestor = interface
end;
IDescendant = interface(IAncestor)
    procedure P1;
end;
TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
    procedure P2;
end;
    .
    .
    .
var
    D: IDescendant;
    A: IAncestor;
begin
    D := TSomething.Create;  // works!
    A := TSomething.Create;  // error
    D.P1;  // works!
    D.P2;  // error
end;
```

In this example, A is declared as a variable of type `IAncestor`. Because `TSomething` does not list IAncestor among the interfaces it implements, a `TSomething` instance cannot be assigned to A. But if we changed `TSomething`'s declaration to

```
TSomething = class(TInterfacedObject, IAncestor, IDescendant)
.
.
.
```

the first error would become a valid assignment. D is declared as a variable of type `IDescendant`. While D references an instance of `TSomething`, we cannot use it to access `TSomething's` P2 method, since P2 is not a method of `IDescendant`. But if we changed D's declaration to

```
D: TSomething;
```

the second error would become a valid method call.

On the Win32 platform, interface references are typically managed through reference-counting, which depends on the `_AddRef` and `_Release` methods inherited from IInterface. These methods, and reference counting in general, are not applicable on the .NET platform, which is a garbage collected environment. Using the default implementation of reference counting, when an object is referenced only through interfaces, there is no need to destroy it manually; the object is automatically destroyed when the last reference to it goes out of scope. Some classes implement interfaces to bypass this default lifetime management, and some hybrid objects use reference counting only when the object does not have an owner.

Global interface-type variables can be initialized only to nil.

To determine whether an interface-type expression references an object, pass it to the standard function Assigned.

Interface Assignment Compatibility

Variables of a given class type are assignment-compatible with any interface type implemented by the class. Variables of an interface type are assignment-compatible with any ancestor interface type. The value nil can be assigned to any interface-type variable.

An interface-type expression can be assigned to a variant. If the interface is of type `IDispatch` or a descendant, the variant receives the type code `varDispatch`. Otherwise, the variant receives the type code `varUnknown`.

A variant whose type code is `varEmpty`, `varUnknown`, or `varDispatch` can be assigned to an `IInterface` variable. A variant whose type code is varEmpty or varDispatch can be assigned to an `IDispatch` variable.

Interface Typecasts

An interface-type expression can be cast to Variant. If the interface is of type `IDispatch` or a descendant, the resulting variant has the type code `varDispatch`. Otherwise, the resulting variant has the type code `varUnknown`.

A variant whose type code is `varEmpty`, `varUnknown`, or `varDispatch` can be cast to `IInterface`. A variant whose type code is `varEmpty` or `varDispatch` can be cast to `IDispatch`.

Interface Querying

You can use the as operator to perform checked interface typecasts. This is known as interface querying, and it yields an interface-type expression from an object reference or from another interface reference, based on the actual (runtime) type of the object. An interface query has the form

```
object as interface
```

where object is an expression of an interface or variant type or denotes an instance of a class that implements an interface, and interface is any interface declared with a GUID.

An interface query returns nil if object is nil. Otherwise, it passes the GUID of interface to the `QueryInterface` method in object, raising an exception unless `QueryInterface` returns zero. If `QueryInterface` returns zero (indicating that object's class implements interface), the interface query returns an interface reference to object.

Automation Objects (Win32 Only)

An object whose class implements the IDispatch interface (declared in the `System` unit) is an Automation object.

Use variants to access Automation objects. When a variant references an Automation object, you can call the object's methods and read or write to its properties through the variant. To do this, you must include `ComObj` in the uses clause of one of your units or your program or library.

Dispatch Interface Types

Dispatch interface types define the methods and properties that an Automation object implements through IDispatch. Calls to methods of a dispatch interface are routed through IDispatch's Invoke method at runtime; a class cannot implement a dispatch interface.

A dispatch interface type declaration has the form

```
type interfaceName = dispinterface
    ['{GUID}']
    memberList
end;
```

where ['{GUID}'] is optional and memberList consists of property and method declarations. Dispatch interface declarations are similar to regular interface declarations, but they cannot specify an ancestor. For example,

```
type
         IStringsDisp = dispinterface
         ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
         property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
         function Count: Integer; dispid 1;
         property Item[Index: Integer]: OleVariant dispid 2;
         procedure Remove(Index: Integer); dispid 3;
         procedure Clear; dispid 4;
         function Add(Item: OleVariant): Integer; dispid 5;
            function _NewEnum: IUnknown; dispid -4;
     end;
```

Dispatch interface methods

Methods of a dispatch interface are prototypes for calls to the Invoke method of the underlying IDispatch implementation. To specify an Automation dispatch ID for a method, include the dispid directive in its declaration, followed by an integer constant; specifying an already used ID causes an error.

A method declared in a dispatch interface cannot contain directives other than dispid. Parameter and result types must be automatable. In other words, they must be Byte, Currency, Real, Double, Longint, Integer, Single, Smallint, AnsiString, WideString, TDateTime, Variant, OleVariant, WordBool, or any interface type.

Dispatch interface properties

Properties of a dispatch interface do not include access specifiers. They can be declared as read only or write only. To specify a dispatch ID for a property, include the dispid directive in its declaration, followed by an integer constant; specifying an already used ID causes an error. Array properties can be declared as default. No other directives are allowed in dispatch-interface property declarations.

Accessing Automation Objects

Automation object method calls are bound at runtime and require no previous method declarations. The validity of these calls is not checked at compile time.

The following example illustrates Automation method calls. The `CreateOleObject` function (defined in `ComObj`) returns an `IDispatch` reference to an Automation object and is assignment-compatible with the variant Word.

```
var
 Word: Variant;
  begin
   Word := CreateOleObject('Word.Basic');
   Word.FileNew('Normal');
   Word.Insert('This is the first line'#13);
   Word.Insert('This is the second line'#13);
   Word.FileSaveAs('c:\temp\test.txt', 3);
  end;
```

You can pass interface-type parameters to Automation methods.

Variant arrays with an element type of `varByte` are the preferred method of passing binary data between Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the `VarArrayLock` and `VarArrayUnlock` routines.

Automation Object Method-Call Syntax

The syntax of an Automation object method call or property access is similar to that of a normal method call or property access. Automation method calls, however, can use both positional and named parameters. (But some Automation servers do not support named parameters.)

A positional parameter is simply an expression. A named parameter consists of a parameter identifier, followed by the **:=** symbol, followed by an expression. Positional parameters must precede any named parameters in a method call. Named parameters can be specified in any order.

Some Automation servers allow you to omit parameters from a method call, accepting their default values. For example,

```
    Word.FileSaveAs('test.doc');
    Word.FileSaveAs('test.doc', 6);
    Word.FileSaveAs('test.doc',,,'secret');
    Word.FileSaveAs('test.doc', Password := 'secret');
    Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Automation method call parameters can be of integer, real, string, Boolean, and variant types. A parameter is passed by reference if the parameter expression consists only of a variable reference, and if the variable reference is of type Byte, Smallint, Integer, Single, Double, Currency, TDateTime, AnsiString, WordBool, or Variant. If the expression is not of one of these types, or if it is not just a variable, the parameter is passed by value. Passing a parameter by reference to a method that expects a value parameter causes COM to fetch the value from the reference parameter. Passing a parameter by value to a method that expects a reference parameter causes an error.

Dual Interfaces

A dual interface is an interface that supports both compile-time binding and runtime binding through Automation. Dual interfaces must descend from IDispatch.

All methods of a dual interface (except from those inherited from `IInterface` and `IDispatch`) must use the safecall convention, and all method parameter and result types must be automatable. (The automatable types are Byte, Currency, Real, Double, Real48, Integer, Single, Smallint, AnsiString, ShortString, TDateTime, Variant, OleVariant, and WordBool.)

Memory Management

This section describes memory management issues related to programming in Delphi on Win32, and on .NET.

Memory Management on the Win32 Platform

The following material describes how memory management on Win32 is handled, and briefly describes memory issues of variables.

The Memory Manager (Win32 Only)

The memory manager manages all dynamic memory allocations and deallocations in an application. The `New`, `Dispose`, `GetMem`, `ReallocMem`, and `FreeMem` standard procedures use the memory manager, and all objects and long strings are allocated through the memory manager.

The memory manager is optimized for applications that allocate large numbers of small- to medium-sized blocks, as is typical for object-oriented applications and applications that process string data. Other memory managers, such as the implementations of `GlobalAlloc`, `LocalAlloc`, and private heap support in Windows, typically do not perform well in such situations, and would slow down an application if they were used directly.

To ensure the best performance, the memory manager interfaces directly with the Win32 virtual memory API (the `VirtualAlloc` and `VirtualFree` functions). The memory manager reserves memory from the operating system in 1Mb sections of address space, and commits memory as required in 16K increments. It decommits and releases unused memory in 16K and 1Mb sections. For smaller blocks, committed memory is further suballocated.

Memory manager blocks are always rounded upward to a 4-byte boundary, and always include a 4-byte header in which the size of the block and other status bits are stored. This means that memory manager blocks are always double-word-aligned, which guarantees optimal CPU performance when addressing the block.

The memory manager maintains two status variables, `AllocMemCount` and `AllocMemSize`, which contain the number of currently allocated memory blocks and the combined size of all currently allocated memory blocks. Applications can use these variables to display status information for debugging.

The `System` unit provides two procedures, `GetMemoryManager` and `SetMemoryManager`, that allow applications to intercept low-level memory manager calls. The `System` unit also provides a function called `GetHeapStatus` that returns a record containing detailed memory-manager status information.

Variables

Global variables are allocated on the application data segment and persist for the duration of the program. Local variables (declared within procedures and functions) reside in an application's stack. Each time a procedure or function is called, it allocates a set of local variables; on exit, the local variables are disposed of. Compiler optimization may eliminate variables earlier.

On Win32, an application's stack is defined by two values: the minimum stack size and the maximum stack size. The values are controlled through the `$MINSTACKSIZE` and `$MAXSTACKSIZE` compiler directives, and default to 16,384 (16K) and 1,048,576 (1Mb) respectively. An application is guaranteed to have the minimum stack size available, and an application's stack is never allowed to grow larger than the maximum stack size. If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

If a Win32 application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated in 4K increments. If allocation of additional stack space fails, either because more memory is not available or because the total size of the stack would exceed the maximum stack size, an EStackOverflow exception is raised. (Stack overflow checking is completely automatic. The `$S` compiler directive, which originally controlled overflow checking, is maintained for backward compatibility.)

Dynamic variables created with the `GetMem` or `New` procedure are heap-allocated and persist until they are deallocated with `FreeMem` or `Dispose`.

Long strings, wide strings, dynamic arrays, variants, and interfaces are heap-allocated, but their memory is managed automatically.

Internal Data Formats

The following topics describe the internal formats of Delphi data types.

Integer Types

The format of an integer-type variable depends on its minimum and maximum bounds.

- If both bounds are within the range 128..127 (Shortint), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (Byte), the variable is stored as an unsigned byte.
- If both bounds are within the range 32768..32767 (Smallint), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (Word), the variable is stored as an unsigned word.
- If both bounds are within the range 2147483648..2147483647 (Longint), the variable is stored as a signed double word.
- If both bounds are within the range 0..4294967295 (Longword), the variable is stored as an unsigned double word.
- Otherwise, the variable is stored as a signed quadruple word (Int64).

Note: a "word" occupies two bytes.

Character Types

On the Win32 platform, Char, an AnsiChar, or a subrange of a Char type is stored as an unsigned byte. A WideChar is stored as an unsigned word.

On the .NET platform, a Char is equivalent to WideChar.

Boolean Types

A Boolean type is stored as a Byte, a ByteBool is stored as a Byte, a WordBool type is stored as a Word, and a LongBool is stored as a Longint.

A Boolean can assume the values 0 (False) and 1 (True). ByteBool, WordBool, and LongBool types can assume the values 0 (False) or nonzero (True).

Enumerated Types

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values and the type was declared in the `{$Z1}` state (the default). If an enumerated type has more than 256 values, or if the type was declared in the `{$Z2}` state, it is stored as an unsigned word. If an enumerated type is declared in the `{$Z4}` state, it is stored as an unsigned double-word.

Real Types

The real types store the binary representation of a sign (**+** or **-**), an exponent, and a significand. A real value has the form

+/- *significand* * 2exponent

where the *significand* has a single bit to the left of the binary decimal point. (That is, 0 <= *significand* < 2.)

In the figures that follow, the most significant bit is always on the left and the least significant bit on the right. The numbers at the top indicate the width (in bits) of each field, with the leftmost items stored at the highest addresses.

For example, for a Real48 value, e is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real48 type

The following discussion of the Real48 type applies only to the Win32 platform. The Real48 type is not supported on the .NET platform.

On the Win32 platform, a 6-byte (48-bit) Real48 number is divided into three fields:

1	39	8
s	f	e

If 0 < *e* <= 255, the value *v* of the number is given by

v = (1)^s * 2^(e129) * (1.f)

If e = 0, then v = 0.

The Real48 type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a Real48, while NaNs and infinities produce an overflow error if an attempt is made to store them in a Real48.

The Single type

A 4-byte (32-bit) Single number is divided into three fields

1	8	23
s	e	f

The value *v* of the number is given by

if 0 < e < 255, then v = (1)^s * 2^(e127) * (1.f)

if e = 0 and f <> 0, then v = (1)^s * 2^(126) * (0.f)

if e = 0 and f = 0, then v = (1)^s * 0

if e = 255 and f = 0, then v = (1)^s * Inf

if e = 255 and f <> 0, then v is a NaN

The Double type

An 8-byte (64-bit) Double number is divided into three fields

1	11	52
s	e	f

The value *v* of the number is given by

if 0 < e < 2047, then v = (1)^s * 2^(e1023) * (1.f)

if e = 0 and f <> 0, then v = (1)^s * 2^(1022) * (0.f)

if e = 0 and f = 0, then v = (1)^s * 0

if e = 2047 and f = 0, then v = (1)^s * Inf

if e = 2047 and f <> 0, then v is a NaN

The Extended type

A 10-byte (80-bit) Extended number is divided into four fields:

1	15	1	63
s	e	i	f

The value *v* of the number is given by

if 0 <= e < 32767, then v = (1)^s * 2^(e16383) * (i.f)

if e = 32767 and f = 0, then v = (1)^s * Inf

if e = 32767 and f <> 0, then v is a NaN

Note: On the .NET platform, the Extended type is aliased to Double, and has been deprecated.

The Comp type

An 8-byte (64-bit) Comp number is stored as a signed 64-bit integer.

Note: On the .NET platform, the Comp type is aliased to Int64, and has been deprecated.

The Currency type

An 8-byte (64-bit) Currency number is stored as a scaled and signed 64-bit integer with the four least-significant digits implicitly representing four decimal places.

Pointer Types

A Pointer type is stored in 4 bytes as a 32-bit address. The pointer value nil is stored as zero.

Note: On the .NET platform, the size of a pointer will vary at runtime. Therefore, `SizeOf(pointer)` is not a compile-time constant, as it is on the Win32 platform.

Short String Types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (`string[255]`).

Note: On the .NET platform, the short string type is implemented as an array of unsigned bytes.

Long String Types

A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is nil and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a long-string memory block.

Long string dynamic memory layout (Win32 only)

Offset	Contents
-8	32-bit reference-count
-4	length in bytes
0..Length - 1	character string
Length	NULL character

The NULL character at the end of a long string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a long string directly to a null-terminated string.

For string constants and literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of -1. When a long string variable is assigned a string constant, the string pointer is assigned the address of the memory block generated for the string constant. The built-in string handling routines know not to attempt to modify blocks that have a reference count of -1.

Note: On the .NET platform, an AnsiString is implemented as an array of unsigned bytes. The information above on string constants and literals does not apply to the .NET platform.

Wide String Types

On Win32, a wide string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a wide string variable is empty (contains a zero-length string), the string pointer is nil and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator. The table below shows the layout of a wide string memory block on Windows.

Wide string dynamic memory layout (Win32 only)

Offset	Contents
-4	32-bit length indicator (in bytes)
0..Length -1	character string
Length	NULL character

The string length is the number of bytes, so it is twice the number of wide characters contained in the string.

The NULL character at the end of a wide string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a wide string directly to a null-terminated string.

On the .NET platform, String and WideString types are implemented using the System.String type. An empty string is not a nil pointer, and the string data is not terminated with a null character.

Set Types

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is equal to

(*Max* div 8) (*Min* div 8) + 1

where *Max* and *Min* are the upper and lower bounds of the base type of the set. The byte number of a specific element *E* is

(*E* div 8) (*Min* div 8)

and the bit number within that byte is

E mod 8

where E denotes the ordinal value of the element. When possible, the compiler stores sets in CPU registers, but a set always resides in memory if it is larger than the generic Integer type or if the program contains code that takes the address of the set.

Note: On the .NET platform, sets containing more than 32 elements are implemented as an array of bytes. The set type in Delphi for .NET is not CLS compliant, therefore other .NET languages cannot use them.

Static Array Types

On the Win32 platform, a static array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

On the .NET platform, static are implemented using the System.Array type. Memory layout is therefore determined by the System.Array type.

Dynamic Array Types

On the Win32 platform, a dynamic-array variable occupies four bytes of memory which contain a pointer to the dynamically allocated array. When the variable is empty (uninitialized) or holds a zero-length array, the pointer is nil and no dynamic memory is associated with the variable. For a nonempty array, the variable points to a dynamically allocated block of memory that contains the array in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a dynamic-array memory block.

Dynamic array memory layout (Win32 only)

Offset	Contents
-8	32-bit reference-count
-4	32-bit length indicator (number of elements)
0..Length * (size of element) -1	array elements

On the .NET platform, dynamic arrays and open array parameters are implemented using the System.Array type. As with static arrays, memory layout is therefore determined by the System.Array type.

Record Types

On the .NET platform, field layout in record types is determined at runtime, and can vary depending on the architecture of the target hardware. The following discussion of record alignment applies to the Win32 platform only (packed records are supported on the .NET platform, however).

When a record type is declared in the `{$A+}` state (the default), and when the declaration does not include a packed modifier, the type is an unpacked record type, and the fields of the record are aligned for efficient access by the CPU. The alignment is controlled by the type of each field and by whether fields are declared together. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, 4, or 8, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. The table below lists the alignments for all data types.

Type alignment masks (Win32 only)

Type	Alignment
Ordinal types	size of the type (1, 2, 4, or 8)
Real types	2 for *Real48*, 4 for *Single*, 8 for *Double* and *Extended*

Short string types	1
Array types	same as the element type of the array.
Record types	the largest alignment of the fields in the record
Set types	size of the type if 1, 2, or 4, otherwise 1
All other types	determined by the $A directive.

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upward to the byte boundary specified by the largest alignment of any of the fields.

If two fields share a common type specification, they are packed even if the declaration does not include the packed modifier and the record type is not declared in the `{$A-}` state. Thus, for example, given the following declaration

```
type
  TMyRecord = record
    A, B: Extended;
    C: Extended;
  end;
```

`A` and `B` are packed (aligned on byte boundaries) because they share the same type specification. The compiler pads the structure with unused bytes to ensure that `C` appears on a quadword boundary.

When a record type is declared in the `{$A-}` state, or when the declaration includes the packed modifier, the fields of the record are not aligned, but are instead assigned consecutive offsets. The total size of such a packed record is simply the size of all the fields. Because data alignment can change, it's a good idea to pack any record structure that you intend to write to disk or pass in memory to another module compiled using a different version of the compiler.

File Types

The following discussion of file types applies to the Win32 platform only. On the .NET platform, text files are implemented with a class (as opposed to a record). Binary file types (e.g. `File of MyType`) are not supported on the .NET platform.

On the Win32 platform, file types are represented as records. Typed files and untyped files occupy 332 bytes, which are laid out as follows:

```
type
TFileRec = packed record
  Handle: Integer;
  Mode: word;
  Flags: word;
    case Byte of
         0: (RecSize: Cardinal);
         1: (BufSize: Cardinal;
               BufPos: Cardinal;
               BufEnd: Cardinal;
               BufPtr: PChar;
               OpenFunc: Pointer;
               InOutFunc: Pointer;
               FlushFunc: Pointer;
               CloseFunc: Pointer;
               UserData: array[1..32] of Byte;
```

```
                Name: array[0..259] of Char; );
end;
```

Text files occupy 460 bytes, which are laid out as follows:

```
type
        TTextBuf = array[0..127] of Char;
        TTextRec = packed record
                Handle: Integer;
                Mode: word;
                Flags: word;
                BufSize: Cardinal;
                BufPos: Cardinal;
                BufEnd: Cardinal;
                BufPtr: PChar;
                OpenFunc: Pointer;
                InOutFunc: Pointer;
                FlushFunc: Pointer;
                CloseFunc: Pointer;
                UserData: array[1..32] of Byte;
                Name: array[0..259] of Char;
                Buffer: TTextBuf;
end;
```

Handle contains the file's handle (when the file is open).

The `Mode` field can assume one of the values

```
const
    fmClosed = $D7B0;
    fmInput= $D7B1;
    fmOutput = $D7B2;
    fmInOut= $D7B3;
```

where *fmClosed* indicates that the file is closed, *fmInput* and *fmOutput* indicate a text file that has been reset (*fmInput*) or rewritten (*fmOutput*), *fmInOut* indicates a typed or untyped file that has been reset or rewritten. Any other value indicates that the file variable is not assigned (and hence not initialized).

The *UserData* field is available for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file; see Device functions. Flags determines the line break style as follows:

bit 0 clear	LF line breaks
bit 0 set	CRLF line breaks

All other Flags bits are reserved for future use.

Procedural Types

On the Win32 platform, a procedure pointer is stored as a 32-bit pointer to the entry point of a procedure or function. A method pointer is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

On the .NET platform, procedural types are implemented using the System.MulticastDelegate class types.

Class Types

The following discussion of the internal layout of class types applies to the Win32 platform only. On the .NET platform, class layout is determined at runtime. Runtime type information is obtained using the System.Reflection APIs in the .NET framework.

On the Win32 platform, a class-type value is stored as a 32-bit pointer to an instance of the class, which is called an *object*. The internal data format of an object resembles that of a record. The object's fields are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the *virtual method table* (VMT) of the class. There is exactly one VMT per class (not one per object); distinct class types, no matter how similar, never share a VMT. VMT's are built automatically by the compiler, and are never directly manipulated by a program. Pointers to VMT's, which are automatically stored by constructor methods in the objects they create, are also never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointersone per user-defined virtual method in the class typein order of declaration. Each slot contains the address of the corresponding virtual method's entry point. This layout is compatible with a C++ v-table and with COM. At negative offsets, a VMT contains a number of fields that are internal to Delphi's implementation. Applications should use the methods defined in TObject to query this information, since the layout is likely to change in future implementations of the Delphi language.

Virtual method table layout (Win32 Only)

Offset	Type	Description
-76	Pointer	pointer to virtual method table (or nil)
-72	Pointer	pointer to interface table (or nil)
-68	Pointer	pointer to Automation information table (or nil)
-64	Pointer	pointer to instance initialization table (or nil)
-60	Pointer	pointer to type information table (or nil)
-56	Pointer	pointer to field definition table (or nil)
-52	Pointer	pointer to method definition table (or nil)
-48	Pointer	pointer to dynamic method table (or nil)
-44	Pointer	pointer to short string containing class name
-40	Cardinal	instance size in bytes
-36	Pointer	pointer to a pointer to ancestor class (or nil)
-32	Pointer	pointer to entry point of *SafecallException* method (or nil)
-28	Pointer	entry point of *AfterConstruction* method
-24	Pointer	entry point of *BeforeDestruction* method
-20	Pointer	entry point of *Dispatch* method
-16	Pointer	entry point of *DefaultHandler* method

-12	Pointer	entry point of *NewInstance* method
-8	Pointer	entry point of *FreeInstance* method
-4	Pointer	entry point of *Destroy* destructor
0	Pointer	entry point of first user-defined virtual method
4	Pointer	entry point of second user-defined virtual method

Class Reference Types

On the Win32 platform, a class-reference value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

On the .NET platform, class reference types are implemented using compiler-constructed nested classes inside the class type they support. These implementation details are subject to change in future compiler releases.

Variant Types

The following discussion of the internal layout of variant types applies to the Win32 platform only. On the .NET platform, variants are an alias of System.Object. Variants rely on boxing and unboxing of data into an object wrapper, as well as Delphi helper classes to implement the variant-related RTL functions.

On the Win32 platform, a variant is stored as a 16-byte record that contains a type code and a value (or a reference to a value) of the type given by the code. The `System` and `Variants` units define constants and types for variants.

The TVarData type represents the internal structure of a Variant variable (on Windows, this is identical to the Variant type used by COM and the Win32 API). The TVarData type can be used in typecasts of Variant variables to access the internal structure of a variable.

The *VType* field of a *TVarData* record contains the type code of the variant in the lower twelve bits (the bits defined by the varTypeMask constant). In addition, the *varArray* bit may be set to indicate that the variant is an array, and the *varByRef* bit may be set to indicate that the variant contains a reference as opposed to a value.

The Reserved1, Reserved2, and Reserved3 fields of a *TVarData* record are unused.

The contents of the remaining eight bytes of a *TVarData* record depend on the *VType* field. If neither the *varArray* nor the *varByRef* bits are set, the variant contains a value of the given type.

If the *varArray* bit is set, the variant contains a pointer to a *TVarArray* structure that defines an array. The type of each array element is given by the *varTypeMask* bits in the *VType* field.

If the *varByRef* bit is set, the variant contains a reference to a value of the type given by the *varTypeMask* and varArray bits in the *VType* field.

The *varString* type code is private. Variants containing a *varString* value should never be passed to a non-Delphi function. On Win32, Delphi's Automation support automatically converts *varString* variants to *varOleStr* variants before passing them as parameters to external functions.

Memory Management Issues on the .NET Platform

The .NET Common Language Runtime is a garbage-collected environment. This means the programmer is freed (for the most part) from worrying about memory allocation and deallocation. Broadly speaking, after you allocate memory, the CLR determines when it is safe to free that memory. "Safe to free" means that no more references to that memory exist.

This topic covers the following memory management issues:

- Creating and destroying objects
- Unit initialization and finalization sections
- Unit initialization and finalization in assemblies and packages

Constructors

In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor.

Note: A constructor can initialize fields from its own class, prior to calling the inherited constructor.

Finalization

Every class in the .NET Framework (including VCL.NET classes) inherits a method called Finalize. The garbage collector calls the Finalize method when the memory for the object is about to be freed. Since the method is called by the garbage collector, you have no control over when it is called. The asynchronous nature of finalization is a problem for objects that open resources such as file handles and database connections, because the Finalize method might not be called for some time, leaving these connections open.

To add a finalizer to a class, override the `strict protected`Finalize procedure that is inherited from TObject. The .NET platform places limits on what you can do in a finalizer, because it is called when the garbage collector is cleaning up objects. The finalizer may execute in a different thread than the thread the object was was created in. A finalizer cannot allocate new memory, and cannot make calls outside of itself. If your class has references to other objects, a finalizer can refer to them (that is, their memory is guaranteed not to have been freed yet), but be aware that their state is undefined, as you do not know whether they have been finalized yet.

When a class has a finalizer, the CLR must add newly instantiated objects of the class to the finalization list. Further, objects with finalizers tend to persist in memory longer, as they are not freed when the garbage collector first determines that they are no longer actively referenced. If the object has references to other objects, those objects are also not freed right away (even if they don’t have finalizers themselves), but must also persist in memory until the original object is finalized. Therefore, finalizers do impart a fair amount of overhead in terms of memory consumption and execution performance, so they should be used judiciously.

It is a good practice to restrict finalizers to small objects that represent unmanaged resources. Classes that use these resources can then hold a reference to the small object with the finalizer. In this way, big classes, and classes that reference many other classes, do not hoard memory because of a finalizer.

Another good practice is to suppress finalizers when a particular resource has already been released in a destructor. After freeing the resources, you can call SuppressFinalize, which causes the CLR to remove the object from the finalization list. Be careful not to call SuppressFinalize with a nil reference, as that causes a runtime exception.

The Dispose Pattern

Another way to free up resources is to implement the *dispose* pattern. Classes adhering to the dispose pattern must implement the .NET interface called IDisposable. IDisposable contains only one method, called Dispose. Unlike the Finalize method, the Dispose method is public. It can be called directly by a user of the class, as opposed to relying on the garbage collector to call it. This gives you back control of freeing resources, but calling Dispose still does not reclaim memory for the object itself - that is still for the garbage collector to do. Note that some classes in the .NET Framework implement both Dispose, and another method such as `Close`. Typically the `Close` method simply calls Dispose, but the extra method is provided because it seems more "natural" for certain classes such as files.

Delphi for .NET classes are free to use the Finalize method for freeing system resources, however the recommended method is to implement the dispose pattern. The Delphi for .NET compiler recognizes a very specific destructor pattern in your class, and implements the IDisposable interface for you. This enables you to continue writing new code for the .NET platform the same way you always have, while allowing much of your existing Win32 Delphi code to run in the garbage collected environment of the CLR.

The compiler recognizes the following specific pattern of a Delphi destructor:

```
TMyClass = class(TObject)
  destructor Destroy; override;
end;
```

Your destructor must fit this pattern exactly:

- The name of the destructor must be `Destroy`.
- The keyword override must be specified.
- The destructor cannot take any parameters.

In the compiler's implementation of the dispose pattern, the Free method is written so that if the class implements the IDisposable interface (which it does), then the Dispose method is called, which in turn calls your destructor.

You can still implement the IDisposable interface directly, if you choose. However, the compiler's automatic implementation of the Free-Dispose-Destroy mechanism cannot coexist with your implementation of IDisposable. The two methods of implementing IDisposable are mutually exclusive. You must choose to either implement IDisposable directly, or equip your class with the familiar `destructor Destroy; override` pattern and rely on the compiler to do the rest. The `Free` method will call Dispose in either case, but if you implement Dispose yourself, you must call your destructor yourself. If you want to implement IDisposable yourself, your destructor cannot be called `Destroy`.

Note: You can declare destructors with other names; the compiler only provides the IDisposable implementation when the destructor fits the above pattern.

The Dispose method is not called automatically; the Free method must be called in order for Dispose to be called. If an object is freed by the garbage collector because there are no references to it, but you did not explicitly call Free on the object, the object will be freed, but the destructor will not execute.

Note: When the garbage collector frees the memory used by an object, it also reclaims the memory used by all fields of the object instance as well. This means the most common reason for implementing destructors in Delphi for Win32 - to release allocated memory - no longer applies. However, in most cases, unmanaged resources such as window handles or file handles still need to be released.

To eliminate the possibility of destructors being called more than once, the Delphi for .NET compiler introduces a field called `DisposeCount` into every class declaration. If the class already has a field by this name, the name collision will cause the compiler to produce a syntax error in the destructor.

Unit Initialization and Finalization

On the .NET platform, units that you depend on will be initialized prior to initializing your own unit. However, there is no way to guarantee the order in which units are initialized. Nor is there a way to guarantee when they will be initialized. Be aware of initialization code that depends on another unit's initialization side effects, such as the creation of a file. Such a dependency cannot be made to work reliably on the .NET platform.

Unit finalization is subject to the same constraints and difficulties as the Finalize method of objects. Specifically, unit finalization is asynchronous, and, there no way to determine when it will happen (or if it will happen, though under most circumstances, it will).

Typical tasks performed in a unit finalization include freeing global objects, unregistering objects that are used by other units, and freeing resources. Because .NET is a memory managed environment, the garbage collector will free global objects even if the unit finalization section is not called. The units in an application domain are loaded and unloaded together, so you do not need to worry about unregistering objects. All units that can possibly refer to each other (even in different assemblies) are released at the same time. Since object references do not cross application domains, there is no danger of something keeping a dangling reference to an object type or code that has been unloaded from memory.

Freeing resources (such as file handles or window handles) is the most important consideration in unit finalization. Because unit finalization sections are not guaranteed to be called, you may want to rewrite your code to handle this issue using finalizers rather than relying on the unit finalization.

The main points to keep in mind for unit initialization and finalization on the .NET platform are:

1. The Finalize method is called asynchronously (both for objects, and for units).
2. Finalization and destructors are used to free unmanaged resources such as file handles. You do not need to destroy object member variables; the garbage collector takes care of this for you.
3. Classes should rely on the compiler's implementation of IDisposable, and provide a destructor called `Destroy`.
4. If a class implements IDisposable itself, it cannot have a destructor called `Destroy`.
5. Reference counting is deprecated. Try to use the `destructor Destroy; override;` pattern wherever possible.
6. Unit initialization should not depend on side effects produced by initialization of dependent units.

Unit Initialization Considerations for Assemblies and Dynamically Linked Packages

Under Win32, the Delphi compiler uses the `DllMain` function as a hook from which to execute unit initialization code. No such execution path exists in the .NET environment. Fortunately, other means of implementing unit initialization exist in the .NET Framework. However, the differences in the implementation between Win32 and .NET could impact the order of unit initialization in your application.

The Delphi for .NET compiler uses CLS-compliant class constructors to implement unit initialization hooks. The CLR requires that every object type have a class constructor. These constructors, or type initializers, are guaranteed to be executed *at most* one time. Class constructors are executed at most one time, because in order for the type to be loaded, it must be used. That is, the assembly containing a type will not be loaded until the type is actually used at runtime. If the assembly is never loaded, its unit initialization section will never run.

Circular unit references also impact the unit initialization process. If unit A uses unit B, and unit B then uses unit A in its implementation section, the order of unit initialization is undefined. To fully understand the possibilities, it is helpful to look at the process one step at a time.

1. Unit A's initialization section uses a type from unit B. If this is the first reference to the type, the CLR will load its assembly, triggering the unit initialization of unit B.
2. As a consequence, loading and initializing unit B occurs before unit A's initialization section has completed execution. Note this is a change from how unit initialization works under Win32.

3. Suppose that unit B's initialization is in progress, and that a type from unit A is used. Unit A has not completed initialization, and such a reference could cause an access violation.

The unit initialization should only use types defined within that unit. Using types from outside the unit will impact unit initialization, and could cause an access violation, as noted above.

Unit initialization for DLLs happens automatically; it is triggered when a type within the DLL is referenced. Applications created with other .NET languages can use Delphi for .NET assemblies without concern for the details of unit initialization.

Program Control

This section describes how parameters are passed to procedures and functions.

Program Control

The concepts of passing parameters and function result processing are important to understand before you undertake your application projects Treatment of parameters and function results is determined by several factors, including calling conventions, parameter semantics, and the type and size of the value being passed.

This following topics are covered in this material:

- Passing Parameters.
- Handling Function Results.
- Handling Method Calls.
- Understanding Exit Procedures.

Passing Parameters

Parameters are transferred to procedures and functions via CPU registers or the stack, depending on the routine's calling convention. For information about calling conventions, see the topic on Calling Conventions.

By Value vs. By Reference

Variable (var) parameters are always passed by reference, as 32-bit pointers that point to the actual storage location.

Value and constant (const) parameters are passed by value or by reference, depending on the type and size of the parameter:

- An ordinal parameter is passed as an 8-bit, 16-bit, 32-bit, or 64-bit value, using the same format as a variable of the corresponding type.
- A real parameter is always passed on the stack. A Single parameter occupies 4 bytes, and a Double, Comp, or Currency parameter occupies 8 bytes. A Real48 occupies 8 bytes, with the Real48 value stored in the lower 6 bytes. An Extended occupies 12 bytes, with the Extended value stored in the lower 10 bytes.
- A short-string parameter is passed as a 32-bit pointer to a short string.
- A long-string or dynamic-array parameter is passed as a 32-bit pointer to the dynamic memory block allocated for the long string. The value **nil** is passed for an empty long string.
- A pointer, class, class-reference, or procedure-pointer parameter is passed as a 32-bit pointer.
- A method pointer is passed on the stack as two 32-bit pointers. The instance pointer is pushed before the method pointer so that the method pointer occupies the lowest address.
- Under the register and pascal conventions, a variant parameter is passed as a 32bit pointer to a Variant value.
- Sets, records, and static arrays of 1, 2, or 4 bytes are passed as 8-bit, 16-bit, and 32bit values. Larger sets, records, and static arrays are passed as 32-bit pointers to the value. An exception to this rule is that records are always passed directly on the stack under the cdecl, stdcall, and safecall conventions; the size of a record passed this way is rounded upward to the nearest double-word boundary.
- An open-array parameter is passed as two 32-bit values. The first value is a pointer to the array data, and the second value is one less than the number of elements in the array.

When two parameters are passed on the stack, each parameter occupies a multiple of 4 bytes (a whole number of double words). For an 8-bit or 16-bit parameter, even though the parameter occupies only a byte or a word, it is passed as a double word. The contents of the unused parts of the double word are undefined.

Pascal, cdecl, stdcall, and safecall Conventions

Under the pascal, cdecl, stdcall and safecall conventions, all parameters are passed on the stack. Under the pascal convention, parameters are pushed in the order of their declaration (left-to-right), so that the first parameter ends up

at the highest address and the last parameter ends up at the lowest address. Under the cdecl, stdcall, and safecall conventions, parameters are pushed in reverse order of declaration (right-to-left), so that the first parameter ends up at the lowest address and the last parameter ends up at the highest address.

Register Convention

Under the register convention, up to three parameters are passed in CPU registers, and the rest (if any) are passed on the stack. The parameters are passed in order of declaration (as with the pascal convention), and the first three parameters that qualify are passed in the EAX, EDX, and ECX registers, in that order. Real, method-pointer, variant, Int64, and structured types do not qualify as register parameters, but all other parameters do. If more than three parameters qualify as register parameters, the first three are passed in EAX, EDX, and ECX, and the remaining parameters are pushed onto the stack in order of declaration. For example, given the declaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

a call to Test passes A in EAX as a 32-bit integer, B in EDX as a pointer to a Char, and D in ECX as a pointer to a long-string memory block; C and E are pushed onto the stack as two double-words and a 32-bit pointer, in that order.

Register saving conventions

Procedures and functions must preserve the EBX, ESI, EDI, and EBP registers, but can modify the EAX, EDX, and ECX registers. When implementing a constructor or destructor in assembler, be sure to preserve the DL register. Procedures and functions are invoked with the assumption that the CPU's direction flag is cleared (corresponding to a CLD instruction) and must return with the direction flag cleared.

Note: Delphi language procedures and functions are generally invoked with the assumption that the FPU stack is empty: The compiler tries to use all eight FPU stack entries when it generates code.

When working with the MMX and XMM instructions, be sure to preserve the values of the xmm and mm registers. Delphi functions are invoked with the assumption that the x87 FPU data registers are available for use by x87 floating point instructions. That is, the compiler assumes that the EMMS/FEMMS instruction has been called after MMX operations. Delphi functions do not make any assumptions about the state and content of xmm registers. They do not guarantee that the content of xmm registers is unchanged.

Handling Function Results

The following conventions are used for returning function result values.

- Ordinal results are returned, when possible, in a CPU register. Bytes are returned in AL, words are returned in AX, and double-words are returned in EAX.
- Real results are returned in the floating-point coprocessor's top-of-stack register (ST(0)). For function results of type Currency, the value in ST(0) is scaled by 10000. For example, the Currency value 1.234 is returned in ST (0) as 12340.
- For a string, dynamic array, method pointer, or variant result, the effects are the same as if the function result were declared as an additional var parameter following the declared parameters. In other words, the caller passes an additional 32-bit pointer that points to a variable in which to return the function result.
- Int64 is returned in EDX:EAX.
- Pointer, class, class-reference, and procedure-pointer results are returned in EAX.
- For static-array, record, and set results, if the value occupies one byte it is returned in AL; if the value occupies two bytes it is returned in AX; and if the value occupies four bytes it is returned in EAX. Otherwise, the result is returned in an additional var parameter that is passed to the function after the declared parameters.

Handling Method Calls

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter Self, which is a reference to the instance or class in which the method is called. The Self parameter is passed as a 32-bit pointer.

- Under the register convention, Self behaves as if it were declared before all other parameters. It is therefore always passed in the EAX register.
- Under the pascal convention, Self behaves as if it were declared after all other parameters (including the additional var parameter sometimes passed for a function result). It is therefore pushed last, ending up at a lower address than all other parameters.
- Under the cdecl, stdcall, and safecall conventions, Self behaves as if it were declared before all other parameters, but after the additional var parameter (if any) passed for a function result. It is therefore the last to be pushed, except for the additional var parameter.

Constructors and destructors use the same calling conventions as other methods, except that an additional Boolean flag parameter is passed to indicate the context of the constructor or destructor call.

A value of False in the flag parameter of a constructor call indicates that the constructor was invoked through an instance object or using the inheritedinherited keyword. In this case, the constructor behaves like an ordinary method. A value of True in the flag parameter of a constructor call indicates that the constructor was invoked through a class reference. In this case, the constructor creates an instance of the class given by Self, and returns a reference to the newly created object in EAX.

A value of False in the flag parameter of a destructor call indicates that the destructor was invoked using the inherited keyword. In this case, the destructor behaves like an ordinary method. A value of True in the flag parameter of a destructor call indicates that the destructor was invoked through an instance object. In this case, the destructor deallocates the instance given by Self just before returning.

The flag parameter behaves as if it were declared before all other parameters. Under the register convention, it is passed in the DL register. Under the pascal convention, it is pushed before all other parameters. Under the cdecl, stdcall, and safecall conventions, it is pushed just before the Self parameter.

Since the DL register indicates whether the constructor or destructor is the outermost in the call stack, you must restore the value of DL before exiting so that BeforeDestruction or AfterConstruction can be called properly.

Understanding Exit Procedures

Exit procedures ensure that specific actionssuch as updating and closing filesare carried out before a program terminates. The `ExitProc` pointer variable allows you to *install* an exit procedure, so that it is always called as part of the program's terminationwhether the termination is normal, forced by a call to Halt, or the result of a runtime error. An exit procedure takes no parameters.

Note: It is recommended that you use finalization sections rather than exit procedures for all exit behavior. `Exit` procedures are available only for executables. For .DLLs (Win32) you can use a similar variable, *DllProc*, which is called when the library is loaded as well as when it is unloaded. For packages, exit behavior must be implemented in a finalization section. All exit procedures are called before execution of finalization sections.

Units as well as programs can install exit procedures. A unit can install an exit procedure as part of its initialization code, relying on the procedure to close files or perform other clean-up tasks.

When implemented properly, an exit procedure is part of a chain of exit procedures. The procedures are executed in reverse order of installation, ensuring that the exit code of one unit isn't executed before the exit code of any units that depend on it. To keep the chain intact, you must save the current contents of `ExitProc` before pointing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of `ExitProc`.

The following code shows a skeleton implementation of an exit procedure.

```
var
ExitSave: Pointer;

procedure MyExit;

begin
    ExitProc := ExitSave; // always restore old vector first
 .
    .
    .
end;

begin
    ExitSave := ExitProc;
    ExitProc := @MyExit;
    .
    .
    .
end.
```

On entry, the code saves the contents of `ExitProc` in `ExitSave`, then installs the `MyExit` procedure. When called as part of the termination process, the first thing `MyExit` does is reinstall the previous exit procedure.

The termination routine in the runtime library keeps calling exit procedures until `ExitProc` becomes nilnil. To avoid infinite loops, `ExitProc` is set to nil before every call, so the next exit procedure is called only if the current exit procedure assigns an address to `ExitProc`. If an error occurs in an exit procedure, it is not called again.

An exit procedure can learn the cause of termination by examining the `ExitCode` integer variable and the `ErrorAddr` pointer variable. In case of normal termination, `ExitCode` is zero and `ErrorAddr` is nil. In case of termination through a call to `Halt`, `ExitCode` contains the value passed to Halt and ErrorAddr is nil. In case of termination due to a runtime error, ExitCode contains the error code and ErrorAddr contains the address of the invalid statement.

The last exit procedure (the one installed by the runtime library) closes the Input and Output files. If ErrorAddr is not nil, it outputs a runtime error message. To output your own runtime error message, install an exit procedure that examines ErrorAddr and outputs a message if it's not nil; before returning, set ErrorAddr to nil so that the error is not reported again by other exit procedures.

Once the runtime library has called all exit procedures, it returns to the operating system, passing the value stored in ExitCode as a return code.

Inline Assembly Code (Win32 Only)

This section describes the use of the inline assembler on the Win32 platform.

Using Inline Assembly Code (Win32 Only)

The built-in assembler allows you to write assembly code within Delphi programs. The inline assembler is available only on the Win32 Delphi compiler. It has the following features:

- Allows for inline assembly.
- Supports all instructions found in the Intel Pentium 4, Intel MMX extensions, Streaming SIMD Extensions (SSE), and the AMD Athlon (including 3D Now!).
- Provides no macro support, but allows for pure assembly function procedures.
- Permits the use of Delphi identifiers, such as constants, types, and variables in assembly statements.

As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions. See the topic on External declarations for more information. If you have external assembly code that you want to use in your applications, you should consider rewriting it in the Delphi language or minimally reimplement it using the inline assembler.

Using the asm Statement

The built-in assembler is accessed through asm statements, which have the form

```
asm statementList end
```

where *statementList* is a sequence of assembly statements separated by semicolons, end-of-line characters, or Delphi comments.

Comments in an asm statement must be in Delphi style. A semicolon does not indicate that the rest of the line is a comment.

The reserved word inline and the directive assembler are maintained for backward compatibility only. They have no effect on the compiler.

Using Registers

In general, the rules of register use in an asm statement are the same as those of an external procedure or function. An asm statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers. On entry to an asm statement, EBP points to the current stack frame and ESP points to the top of the stack. Except for ESP and EBP, an asm statement can assume nothing about register contents on entry to the statement.

Understanding Assembler Syntax (Win32 Only)

The inline assembler is available only on the Win32 Delphi compiler. The following material describes the elements of the assembler syntax necessary for proper use.

- Assembler Statement Syntax
- Labels
- Instruction Opcodes
- Assembly Directives
- Operands

Assembler Statement Syntax

This syntax of an assembly statement is

```
Label: Prefix Opcode Operand1, Operand2
```

where *Label* is a label, *Prefix* is an assembly prefix opcode (operation code), *Opcode* is an assembly instruction opcode or directive, and *Operand* is an assembly expression. Label and Prefix are optional. Some opcodes take only one operand, and some take none.

Comments are allowed between assembly statements, but not within them. For example,

```
    MOV AX,1 {Initial value}            { OK }
    MOV CX,100 {Count}                                          { OK }

                  MOV {Initial value} AX,1;                   { Error! }
    MOV CX, {Count} 100                                       { Error! }
```

Labels

Labels are used in built-in assembly statements as they are in the Delphi languageby writing the label and a colon before a statement. There is no limit to a label's length. As in Delphi, labels must be declared in a label declaration part in the block containing the asm statement. The one exception to this rule is local labels.

Local labels are labels that start with an at-sign (**@**). They consist of an at-sign followed by one or more letters, digits, underscores, or at-signs. Use of local labels is restricted to asm statements, and the scope of a local label extends from the asm reserved word to the end of the asm statement that contains it. A local label doesn't have to be declared.

Instruction Opcodes

The built-in assembler supports all of the Intel-documented opcodes for general application use. Note that operating system privileged instructions may not be supported. Specifically, the following families of instructions are supported:

- Pentium family
- Pentium Pro and Pentium II
- Pentium III
- Pentium 4

In addition, the built-in assembler supports the following instruction sets

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow! (from the AMD Athlon onwards)

For a complete description of each instruction, refer to your microprocessor documentation.

RET instruction sizing

The RET instruction opcode always generates a near return.

Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient, form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and to all conditional jump instructions when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one-byte opcode followed by a one-byte displacement) if the distance to the target label is 128 to 127 bytes. Otherwise it generates a near jump (one-byte opcode followed by a two-byte displacement).

For a conditional jump instruction, a short jump (one-byte opcode followed by a one-byte displacement) is generated if the distance to the target label is 128 to 127 bytes. Otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (five bytes in total). For example, the assembly statement

```
JC    Stop
```

where Stop isn't within reach of a short jump, is converted to a machine code sequence that corresponds to this:

```
     JNC    Skip
     JMP    Stop
     Skip:
```

Jumps to the entry points of procedures and functions are always near.

Assembly Directives

The built-in assembler supports three assembly define directives: DB (define byte), DW (define word), and DD (define double word). Each generates data corresponding to the comma-separated operands that follow the directive.

The DB directive generates a sequence of bytes. Each operand can be a constant expression with a value between 128 and 255, or a character string of any length. Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.

The DW directive generates a sequence of words. Each operand can be a constant expression with a value between 32,768 and 65,535, or an address expression. For an address expression, the built-in assembler generates a near pointer, a word that contains the offset part of the address.

The DD directive generates a sequence of double words. Each operand can be a constant expression with a value between 2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the built-in assembler generates a far pointer, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

The DQ directive defines a quad word for Int64 values.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembly statements. To generate uninitialized or initialized data in the data segment, you should use Delphi var or const declarations.

Some examples of DB, DW, and DD directives follow.

```
 asm
     DB
FFH                                                                    { One byte }
     DB        0,99
{ Two bytes }
     DB        'A'
{ Ord('A') }
     DB         'Hello world...',0DH,0AH         { String followed by CR/LF }
     DB        12,'string'                                  { Delphi style string }
     DW        0FFFFH                                                 { One word }
     DW         0,9999
{ Two words }
     DW        'A'
{ Same as DB  'A',0 }
     DW         'BA'
{ Same as DB 'A','B' }
     DW         MyVar
{ Offset of MyVar }
     DW         MyProc
{ Offset of MyProc }
     DD        0FFFFFFFFH                                          { One double-word }
     DD        0,999999999                                        { Two double-words }
     DD        'A'
{ Same as DB 'A',0,0,0 }
     DD         'DCBA'
{ Same as DB 'A','B','C','D' }
     DD         MyVar
{ Pointer to MyVar }
     DD         MyProc
{ Pointer to MyProc }
 end;
```

When an identifier precedes a DB, DW , or DD directive, it causes the declaration of a byte-, word-, or double-word-sized variable at the location of the directive. For example, the assembler allows the following:

```
     ByteVar          DB          ?
     WordVar          DW          ?
     IntVar             DD          ?
                      .
                      .
                      .
                MOV          AL,ByteVar
                MOV          BX,WordVar
                MOV             ECX,IntVar
```

The built-in assembler doesn't support such variable declarations. The only kind of symbol that can be defined in an inline assembly statement is a label. All variables must be declared using Delphi syntax; the preceding construction can be replaced by

```
var
```

```
     ByteVar: Byte;
     WordVar: Word;
     IntVar: Integer;
                    .
                    .
                    .

    asm
     MOV AL,ByteVar
     MOV BX,WordVar
     MOV ECX,IntVar
    end;
```

SMALL and LARGE can be used determine the width of a displacement:

```
MOV EAX, [LARGE $1234]
```

This instruction generates a 'normal' move with a 32-bit displacement ($00001234).

```
MOV EAX, [SMALL $1234]
```

The second instruction will generate a move with an address size override prefix and a 16-bit displacement ($1234).

SMALL can be used to save space. The following example generates an address size override and a 2-byte address (in total three bytes)

```
 MOV EAX, [SMALL 123]
```

as opposed to

```
 MOV EAX, [123]
```

which will generate no address size override and a 4-byte address (in total four bytes).

Two additional directives allow assembly code to access dynamic and virtual methods: VMTOFFSET and DMTINDEX.

VMTOFFSET retrieves the offset in bytes of the virtual method pointer table entry of the virtual method argument from the beginning of the virtual method table (VMT). This directive needs a fully specified class name with a method name as a parameter (for example, TExample.VirtualMethod), or an interface name and an interface method name.

DMTINDEX retrieves the dynamic method table index of the passed dynamic method. This directive also needs a fully specified class name with a method name as a parameter, for example, TExample.DynamicMethod. To invoke the dynamic method, call System.@CallDynaInst with the (E)SI register containing the value obtained from DMTINDEX.

Note: Methods with the *message* directive are implemented as dynamic methods and can also be called using the DMTINDEX technique. For example:

```
TMyClass = class
   procedure x; message MYMESSAGE;
end;
```

The following example uses both DMTINDEX and VMTOFFSET to access dynamic and virtual methods:

```
program Project2;
   type
    TExample = class
        procedure DynamicMethod; dynamic;
           procedure VirtualMethod; virtual;
    end;

                procedure TExample.DynamicMethod;
    begin

             end;

                procedure TExample.VirtualMethod;
    begin

             end;

                procedure CallDynamicMethod(e: TExample);
     asm
         // Save ESI register
         PUSH    ESI

                         // Instance pointer needs to be in EAX
         MOV     EAX, e

                         // DMT entry index needs to be in (E)SI
         MOV     ESI, DMTINDEX TExample.DynamicMethod

                         // Now call the method
         CALL    System.@CallDynaInst

                         // Restore ESI register
         POP ESI

                    end;

                    procedure CallVirtualMethod(e: TExample);
     asm
             // Instance pointer needs to be in EAX
             MOV     EAX, e

                             // Retrieve VMT table entry
             MOV     EDX, [EAX]

                             // Now call the method at offset VMTOFFSET
             CALL    DWORD PTR [EDX + VMTOFFSET TExample.VirtualMethod]

                        end;

                    var
         e: TExample;
     begin
         e := TExample.Create;
             try
                 CallDynamicMethod(e);
                 CallVirtualMethod(e);
             finally
                 e.Free;
     end;
```

```
        end.
```

Operands

Inline assembler operands are expressions that consist of constants, registers, symbols, and operators.

Within operands, the following reserved words have predefined meanings:

Built-in assembler reserved words

AH	CL	DX	ESP	mm4	SHL	WORD
AL	CS	EAX	FS	mm5	SHR	xmm0
AND	CX	EBP	GS	mm6	SI	xmm1
AX	DH	EBX	HIGH	mm7	SMALL	xmm2
BH	DI	ECX	LARGE	MOD	SP	xmm3
BL	DL	EDI	LOW	NOT	SS	xmm4
BP	CL	EDX	mm0	OFFSET	ST	xmm5
BX	DMTINDEX	EIP	mm1	OR	TBYTE	xmm6
BYTE	DS	ES	mm2	PTR	TYPE	xmm7
CH	DWORD	ESI	mm3	QWORD	VMTOFFSET	XOR

Reserved words always take precedence over user-defined identifiers. For example,

```
var
  Ch: Char;
        .
        .
        .
asm
  MOV    CH, 1
end;
```

loads 1 into the CH register, not into the *Ch* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (**&**) override operator:

```
MOV&Ch, 1
```

It is best to avoid user-defined identifiers with the same names as built-in assembler reserved words.

Assembly Expressions (Win32 Only)

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants. The inline assembler is available only on the Win32 Delphi compiler.

Expressions are built from expression elements and operators, and each expression has an associated expression class and expression type. This topic covers the following material:

- Differences between Delphi and Assembler Expressions
- Expression Elements
- Expression Classes
- Expression Types
- Expression Operators

Differences between Delphi and Assembler Expressions

The most important difference between Delphi expressions and built-in assembler expressions is that assembler expressions must resolve to a constant value. In other words, it must resolve to a value that can be computed at compile time. For example, given the declarations

```
const
 X = 10;
 Y = 20;
var
 Z: Integer;
```

the following is a valid statement.

```
asm
 MOV        Z,X+Y
end;
```

Because both `X` and `Y` are constants, the expression `X + Y` is a convenient way of writing the constant 30, and the resulting instruction simply moves of the value 30 into the variable `Z`. But if `X` and `Y` are variables

```
var
 X, Y: Integer;
```

the built-in assembler cannot compute the value of `X + Y` at compile time. In this case, to move the sum of `X` and `Y` into `Z` you would use

```
asm
 MOV            EAX,X
 ADD            EAX,Y
 MOV            Z,EAX
end;
```

In a Delphi expression, a variable reference denotes the *contents* of the variable. But in an assembler expression, a variable reference denotes the *address* of the variable. In Delphi the expression `X + 4` (where `X` is a variable)

means the contents of X plus 4, while to the built-in assembler it means the contents of the word at the address four bytes higher than the address of X. So, even though you are allowed to write

```
asm
 MOV            EAX,X+4
end;
```

this code doesn't load the value of X plus 4 into AX; instead, it loads the value of a word stored four bytes beyond X. The correct way to add 4 to the contents of X is

```
asm
 MOV            EAX,X
    ADD            EAX,4
end;
```

Expression Elements

The elements of an expression are constants, registers, and symbols.

Numeric Constants

Numeric constants must be integers, and their values must be between 2,147,483,648 and 4,294,967,295.

By default, numeric constants use decimal notation, but the built-in assembler also supports binary, octal, and hexadecimal. Binary notation is selected by writing a B after the number, octal notation by writing an O after the number, and hexadecimal notation by writing an H after the number or a **$** before the number.

Numeric constants must start with one of the digits 0 through 9 or the **$** character. When you write a hexadecimal constant using the H suffix, an extra zero is required in front of the number if the first significant digit is one of the digits A through F. For example, 0BAD4H and $BAD4 are hexadecimal constants, but BAD4H is an identifier because it starts with a letter.

String Constants

String constants must be enclosed in single or double quotation marks. Two consecutive quotation marks of the same type as the enclosing quotation marks count as only one character. Here are some examples of string constants:

```
'Z'
'Delphi'
'Linux'
"That's all folks"
'"That''s    all folks," he said.'
'100'
'"'
"'"
```

String constants of any length are allowed in DB directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where `Ch1` is the rightmost (last) character and `Ch4` is the leftmost (first) character. If the string is shorter than four characters, the leftmost characters are assumed to be zero. The following table shows string constants and their numeric values.

String examples and their values

String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers in the inline assembler:

CPU registers

32-bit general purpose	EAX EBX ECX EDX	32-bit pointer or index	ESP EBP ESI EDI
16-bit general purpose	AX BX CX DX	16-bit pointer or index	SP BP SI DI
8-bit low registers	AL BL CL DL	16-bit segment registers	CS DS SS ES
		32-bit segment registers	FS GS
8-bit high registers	AH BH CH DH	Coprocessor register stack	ST

When an operand consists solely of a register name, it is called a register operand. All registers can be used as register operands, and some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. You can also index with all the 32-bit registersfor example, [EAX+ECX], [ESP], and [ESP+EAX+5].

The segment registers (ES, CS, SS, DS, FS, and GS) are supported, but segments are normally not useful in 32-bit applications.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(X), where X is a constant between 0 and 7 indicating the distance from the top of the register stack.

Symbols

The built-in assembler allows you to access almost all Delphi identifiers in assembly language expressions, including constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the special symbol @Result, which corresponds to the *Result* variable within the body of a function. For example, the function

```
function Sum(X, Y: Integer): Integer;
begin
        Result := X + Y;
end;
```

could be written in assembly language as

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
 asm
         MOV         EAX,X
            ADD         EAX,Y
            MOV         @Result,EAX
 end;
end;
```

The following symbols cannot be used in asm statements:

- Standard procedures and functions (for example, WriteLn and Chr).
- String, floating-point, and set constants (except when loading registers).
- Labels that aren't declared in the current block.
- The @Result symbol outside of functions.

The following table summarizes the kinds of symbol that can be used in asm statements.

Symbols recognized by the built-in assembler

Symbol	Value	Class	Type
Label	Address of label	Memory reference	Size of type
Constant	Value of constant	Immediate value	0
Type	0	Memory reference	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable or address of a pointer to the variable	Memory reference	Size of type
Procedure	Address of procedure	Memory reference	Size of type
Function	Address of function	Memory reference	Size of type
Unit	0	Immediate value	0
@Result	Result variable offset	Memory reference	Size of type

With optimizations disabled, local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP, and the value of a local variable symbol is its signed offset from EBP. The assembler automatically adds [EBP] in references to local variables. For example, given the declaration

```
var Count: Integer;
```

within a function or procedure, the instruction

```
MOV         EAX,Count
```

assembles into `MOV EAX,[EBP4]`.

The built-in assembler treats var parameters as a 32-bit pointers, and the size of a var parameter is always 4. The syntax for accessing a var parameter is different from that for accessing a value parameter. To access the contents of a var parameter, you must first load the 32-bit pointer and then access the location it points to. For example,

```
function Sum(var X, Y: Integer): Integer; stdcall;
    begin
         asm
                 MOV                EAX,X
                 MOV                EAX,[EAX]
                 MOV                EDX,Y
                 ADD                EAX,[EDX]
                 MOV                @Result,EAX
     end;
    end;
```

Identifiers can be qualified within asm statements. For example, given the declarations

```
 type
     TPoint = record
             X, Y: Integer;
     end;
     TRect = record
             A, B: TPoint;
     end;
    var
     P: TPoint;
     R: TRect;
```

the following constructions can be used in an asm statement to access fields.

```
MOV             EAX,P.X
MOV             EDX,P.Y
MOV             ECX,R.A.X
MOV             EBX,R.B.Y
```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of [EDX] into EAX.

```
MOV             EAX,(TRect PTR [EDX]).B.X
MOV             EAX,TRect([EDX]).B.X
MOV             EAX,TRect[EDX].B.X
MOV             EAX,[EDX].TRect.B.X
```

Expression Classes

The built-in assembler divides expressions into three classes: registers, memory references, and immediate values.

An expression that consists solely of a register name is a register expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references. Delphi's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values. This group includes Delphi's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
 Start = 10;
var
 Count: Integer;
    .
    .
    .
asm
 MOV            EAX,Start                                    { MOV EAX,xxxx }
 MOV            EBX,Count                                    { MOV EBX,[xxxx] }
 MOV            ECX,[Start]                            { MOV ECX,[xxxx] }
 MOV            EDX,OFFSET Count            { MOV EDX,xxxx }
end;
```

Because Start is an immediate value, the first MOV is assembled into a move immediate instruction. The second MOV, however, is translated into a move memory instruction, as Count is a memory reference. In the third MOV, the brackets convert Start into a memory reference (in this case, the word at offset 10 in the data segment). In the fourth MOV, the OFFSET operator converts Count into an immediate value (the offset of Count in the data segment).

The brackets and OFFSET operator complement each other. The following asm statement produces identical machine code to the first two lines of the previous asm statement.

```
asm
 MOV            EAX,OFFSET [Start]
 MOV            EBX,[OFFSET Count]
end;
```

Memory references and immediate values are further classified as either relocatable or absolute. Relocation is the process by which the linker assigns absolute addresses to symbols. A relocatable expression denotes a value that requires relocation at link time, while an absolute expression denotes a value that requires no such relocation. Typically, expressions that refer to labels, variables, procedures, or functions are relocatable, since the final address of these symbols is unknown at compile time. Expressions that operate solely on constants are absolute.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Expression Types

Every built-in assembler expression has a typeor, more correctly, a size, because the assembler regards the type of an expression simply as the size of its memory location. For example, the type of an Integer variable is four, because it occupies 4 bytes. The built-in assembler performs type checking whenever possible, so in the instructions

```
var
 QuitFlag: Boolean;
 OutBufPtr: Word;
 .
    .
    .
asm
```

```
 MOV             AL,QuitFlag
 MOV             BX,OutBufPtr
end;
```

the assembler checks that the size of `QuitFlag` is one (a byte), and that the size of `OutBufPtr` is two (a word). The instruction

```
MOV             DL,OutBufPtr
```

produces an error because DL is a byte-sized register and `OutBufPtr` is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
MOV             DL,BYTE PTR OutBufPtr
MOV             DL,Byte(OutBufPtr)
MOV             DL,OutBufPtr.Byte
```

These MOV instructions all refer to the first (least significant) byte of the `OutBufPtr` variable.

In some cases, a memory reference is untyped. One example is an immediate value (Buffer) enclosed in square brackets:

```
procedure Example(var Buffer);
     asm
         MOV AL,     [Buffer]
         MOV CX,     [Buffer]
         MOV EDX, [Buffer]
     end;
```

The built-in assembler permits these instructions, because the expression [Buffer] has no typeit just means "the contents of the location indicated by Buffer," and the type can be determined from the first operand (byte for AL, word for CX, and double-word for EDX.

In cases where the type can't be determined from another operand, the built-in assembler requires an explicit typecast. For example,

```
INC     BYTE PTR [ECX]
IMUL    WORD PTR [EDX]
```

The following table summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Delphi types.

Predefined type symbols

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Expression Operators

The built-in assembler provides a variety of operators. Precedence rules are different from that of the Delphi language; for example, in an asm statement, AND has lower precedence than the addition and subtraction operators. The following table lists the built-in assembler's expression operators in decreasing order of precedence.

Precedence of built-in assembler expression operators

Operators	Remarks	Precedence
&		highest
(...), **[...]**,., HIGH, LOW		
+, **-**	unary **+** and **-**	
:		
OFFSET, TYPE, PTR, *, /, MOD, SHL, SHR, **+**, **-**	binary **+** and **-**	
NOT, AND, OR, XOR		lowest

The following table defines the built-in assembler's expression operators.

Definitions of built-in assembler expression operators

Operator	Description
&	**Identifier override**. The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(...)	**Subexpression**. Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the parentheses; the result in this case is the sum of the values of the two expressions, with the type of the first expression.
[...]	**Memory reference**. The expression within brackets is evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the brackets; the result in this case is the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	**Structure member selector**. The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	**Unary plus**. Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
-	**Unary minus**. Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
+	**Addition**. The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions is a memory reference, the result is also a memory reference.
-	**Subtraction**. The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
:	**Segment override**. Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction is prefixed with an appropriate segment-override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (double word) of the expression following the operator. The result is an immediate value.

TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	**Typecast operator**. The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	**Multiplication**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	**Integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	**Remainder after integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	**Logical shift left**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	**Logical shift right**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
NOT	**Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	**Bitwise AND**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	**Bitwise OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	**Bitwise exclusive OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Assembly Procedures and Functions (Win32 Only)

You can write complete procedures and functions using inline assembly language code, without including a `begin...end` statement. This topic covers these issues:

- Compiler Optimizations.
- Function Results.

The inline assembler is available only on the Win32 Delphi compiler.

Compiler Optimizations

An example of the type of function you can write is as follows:

```
function LongMul(X, Y: Integer): Longint;
  asm
       MOV        EAX,X
                IMUL Y
        end;
```

The compiler performs several optimizations on these routines:

- No code is generated to copy value parameters into local variables. This affects all string-type value parameters and other value parameters whose size isn't 1, 2, or 4 bytes. Within the routine, such parameters must be treated as if they were var parameters.
- Unless a function returns a string, variant, or interface reference, the compiler doesn't allocate a function result variable; a reference to the @Result symbol is an error. For strings, variants, and interfaces, the caller always allocates an @Result pointer.
- The compiler only generates stack frames for nested routines, for routines that have local parameters, or for routines that have parameters on the stack.
- Locals is the size of the local variables and Params is the size of the parameters. If both Locals and Params are zero, there is no entry code, and the exit code consists simply of a RET instruction.

The automatically generated entry and exit code for the routine looks like this:

```
PUSH            EBP                                 ;Present if Locals <> 0 or Params <> 0
MOV               EBP,ESP                   ;Present if Locals <> 0 or Params <> 0
SUB               ESP,Locals        ;Present if Locals <> 0
.
.
.
MOV               ESP,EBP                   ;Present if Locals <> 0
POP              EBP                                 ;Present if Locals <> 0 or Params <> 0
RET               Params                          ;Always present
```

If locals include variants, long strings, or interfaces, they are initialized to zero but not finalized.

Function Results

Assembly language functions return their results as follows.

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real values are returned in ST(0) on the coprocessor's register stack. (Currency values are scaled by 10000.)

- Pointers, including long strings, are returned in EAX.
- Short strings and variants are returned in the temporary location pointed to by `@Result`.

.NET Topics

This section contains information specific to programming in Delphi on the .NET platform.

Using .NET Custom Attributes

.NET framework assemblies are self-describing entities. They contain intermediate code that is compiled to native machine instructions when the assembly is loaded. More than that, assemblies contain a wealth of information about that code. The compiler emits this descriptive information, or metadata, into the assembly as it processes the source code. In other programming environments, there is no way to access metadata once your code is compiled; the information is lost during the compilation process. On the .NET platform, however, you have the ability to access metadata using runtime reflection services.

The .NET framework gives you the ability to extend the metadata emitted by the compiler with your own descriptive attributes. These customized attributes are somewhat analogous to language keywords, and are stored with the other metadata in the assembly.

- Declaring custom attributes
- Using custom attributes
- Custom attributes and interfaces

Declaring a Custom Attribute Class

Creating a custom attribute is the same as declaring a class. The custom attribute class has a constructor, and properties to set and retrieve its state data. Custom attributes must inherit from TCustomAttribute. The following code declares a custom attribute with a constructor and two properties:

```
type
   TCustomCodeAttribute = class(TCustomAttribute)
      private
          Fprop1 : integer;
          Fprop2 : integer;
          aVal   : integer;
          procedure Setprop1(p1 : integer);
          procedure Setprop2(p2 : integer);
      public
          constructor Create(const myVal : integer);
          property prop1 : integer read Fprop1 write Setprop1;
          property prop2 : integer read Fprop2 write Setprop2;
   end;
```

The implementation of the constructor might look like

```
constructor TCustomCodeAttribute.Create(const myVal : integer);
begin
    inherited Create;
    aVal := myVal;
end;
```

Delphi for .NET supports the creation of custom attribute classes, as shown above, and all of the custom attributes provided by the .NET framework.

Using Custom Attributes

Custom attributes are placed directly before the source code symbol to which the attribute applies. Attributes can be placed before

- variables and constants
- procedures and functions
- function results
- procedure and function parameters
- types
- fields, properties, and methods

Note that Delphi for .NET supports the use of named properties in the initialization. These can be the names of properties, or of public fields of the custom attribute class. Named properties are listed after all of the parameters required by the constructor. For example

```
[TCustomCodeAttribute(1024, prop1=512, prop2=128)]
TMyClass = class(TObject)
...
end;
```

applies the custom attribute declared above to the class `TMyClass`.

The first parameter, 1024, is the value required by the constructor. The second two parameters are the properties defined in the custom attribute.

When a custom attribute is placed before a list of multiple variable declarations, the attribute applies to all variables declared in that list. For example

```
var
 [TCustomAttribute(1024, prop1=512, prop2=128)]
 x, y, z: Integer;
```

would result in `TCustomAttribute` being applied to all three variables, `X`, `Y`, and `Z`.

Custom attributes applied to types can be detected at runtime with the GetCustomAttributes method of the Type class. The following Delphi code demonstrates how to query for custom attributes at runtime.

```
var
   F: TMyClass;          // TMyClass declared above
   T: System.Type;
   A: array of TObject;  // Will hold custom attributes
   I: Integer;

begin
   F := TMyClass.Create;
   T := F.GetType;
   A := T.GetCustomAttributes(True);

   // Write the type name, and then loop over custom
   // attributes returned from the call to
   // System.Type.GetCustomAttributes.
   Writeln(T.FullName);
   for I := Low(A) to High(A) do
      Writeln(A[I].GetType.FullName);
end.
```

Using the DllImport Custom Attribute

You can call unmanaged Win32 APIs (and other unmanaged code) by prefixing the function declaration with the `DllImport` custom attribute. This attribute resides in the `System.Runtime.InteropServices` namespace, as shown below:

```
Program HellowWorld2;

    // Don't forget to include the InteropServices unit when using the DllImport attribute.
     uses System.Runtime.InteropServices;

     [DllImport('user32.dll')]
     function MessageBeep(uType : LongWord) : Boolean; external;

     begin
           MessageBeep(LongWord(-1));
     end.
```

Note the external keyword is still required, to replace the block in the function declaration. All other attributes, such as the calling convention, can be passed through the `DllImport` custom attribute.

Custom Attributes and Interfaces

Delphi syntax dictates that the GUID (if present) must immediately follow the declaration of an interface. Since the GUID syntax is similar to that of custom attributes, the compiler must be made to know the difference between a custom attribute - which applies to the next declaration - and a GUID specifier, which applies to the previous declaration. Without this special case, the compiler would try to apply an attribute to the first member of the interface.

When the compiler sees an interface declaration, the next square bracket construct found is assumed to be that of a GUID specifier for the interface. The GUID must be in the traditional Delphi form:

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

Alternatively, you can use the `Guid` custom attribute of the .NET framework (GuidAttribute). If you choose this method, then you should introduce the attribute before the interface, as with any other custom attribute.

The effect in either case is the same: the GUID is emitted into the metadata for the interface type. Note that GUIDs are not required for interfaces in the .NET Framework. They are only used for COM interoperability.

Note: When importing COM interfaces with the `ComImport` custom attribute, you must declare the `GuidAttribute` instead of using the Delphi syntax.

Index