embarcadero®
An Idera, Inc. Company

Product Documentation

Delphi Language Guide

Version 10.3 Rio

Latest version is found on online docwiki.embarcadero.com/RADStudio/en/Delphi_Language_Guide_Index

© 2019 Embarcadero Technologies, Inc. Embarcadero, the Embarcadero Technologies logos, and all other Embarcadero Technologies product or service names are trademarks or registered trademarks of Embarcadero Technologies, Inc. All other trademarks are property of their respective owners.

Embarcadero tools are built for elite developers who build and maintain the world's most critical applications. Our customers choose Embarcadero because we are the champion of developers, and we help them build more secure and scalable enterprise applications faster than any other tools on the market. In fact, ninety of the Fortune 100 and an active community of more than three million users worldwide have relied on Embarcadero's award-winning products for over 30 years.

If you're trying to build a business-critical application in a demanding vertical, Embarcadero is for you. If you're looking to write steadfast code quickly that will pass stringent code reviews faster than any other, Embarcadero is for you. We're here to support elite developers who understand the scalability and stability of C++ and Delphi and depend on the decades of innovation those languages bring to development.

We invite you to try our products for free and see for yourself.
www.embarcadero.com/products/rad-studio/start-for-free

Embarcadero is an Idera, Inc. company. Idera, Inc. is the parent company of global B2B software productivity brands whose solutions enable technical users to do more with less, faster. Idera, Inc. brands span three divisions – Database Tools, Developer Tools, and Test Management Tools – with products that are evangelized by millions of community members and more than 50,000 customers worldwide, including some of the world's largest healthcare, financial services, retail, and technology companies. Embarcadero and Idera are online at www.embarcadero.com and www.ideracorp.com

November, 2019

In support of the education of new developers, and as part of LearnDelphi.org, this document is licensed under a Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)
creativecommons.org/licenses/by-sa/4.0/

Please link back to embarcadero.com and learndelphi.org

LearnDelphi.org

CONTENTS

Language Overview

Delphi is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Based on Delphi, 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 the RAD Studio component framework and environment. For the most part, descriptions and examples in this language guide assume that you are using Embarcadero development tools.

Most developers using Embarcadero software development tools write and compile their code in the integrated development environment (IDE). Embarcadero 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 Delphi language specification. For example, Embarcadero 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 Visual Component Library (VCL). Occasionally, however, Delphi-specific rules are distinguished from rules that apply to all Delphi programming.

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. Embarcadero 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, Embarcadero 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)
- 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 Embarcadero'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) for each new unit used in your project; all the .dcu 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 compiler 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.

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 Embarcadero development tools. First, it is a console application. Embarcadero 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.

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 Vcl.Forms.TApplication class defined in the Forms unit. (Every project has an automatically generated Application object.) Two of these calls invoke a Vcl.Forms.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 Vcl.Forms.TForm) and an instance of this class `Form1`. The `TForm1` class includes a button -- `Button1`, an instance of Vcl.StdCtrls.TButton -- and a procedure named `Button1Click` that is called when the user presses `Button1`. `Button1Click` hides `Form1` and 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. The TForm2 class includes a button (CancelButton, an instance of Vcl.StdCtrls.TButton) and a label (Label1, an instance of Vcl.StdCtrls.TLabel). You can not 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 that will be invoked at run time 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 run time. 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 Index

This chapter provides a more detailed look at Delphi program organization.

Topics

- Programs and Units (Delphi)
- Using Namespaces with Delphi

Programs and Units (Delphi)

This topic covers the overall structure of a Delphi application: the **program** header, **unit** declaration syntax, and the **uses** clause.

- 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.

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. Embarcadero 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.

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. For applications developed using Embarcadero tools, 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);
```

Embarcadero's Delphi ignores these parameters.

In RAD Studio, the **program** heading introduces its own namespace, which is called the project default namespace.

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 Vcl.Forms.TApplication, Web.WebBroker.TWebApplication, or Vcl.SvcMgr.TServiceApplication. The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block. Note that the **end** that represents the end of the program source must be followed by a period (.):

```
begin
  .
  .
  .
end.
```

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...
  // Interface section 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 Embarcadero 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`. 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 **interface uses** 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 older "begin ... end." syntax still functions. Basically, the reserved word "begin" can be used in place of **initialization** followed by zero or more execution statements. Code using the older "begin ... end." syntax cannot specify a **finalization** section. In this case, finalization is accomplished by providing a procedure to the ExitProc variable. This method is not recommended for code going forward, but you might see it used in older source code.

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.

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.

Case Sensitivity: 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` 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

> **Important:** RAD Studio now supports a **unit scope name** or prefix in addition to the namespace or unit name. In order for a name to be considered fully qualified, the unit scope name must be included. For more details, see **Unit Scope Names**.

In Delphi, a *unit* is the basic container for types. In Delphi, a *namespace* is a container of Delphi units.

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.

This topic describes the following:

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

Declaring Namespaces

In RAD Studio, 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 the parts of 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.dcu`.

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 same rule applies for the `dcu` file name. The resulting `dcu` file in the current example would be `MyUnit.dcu`.

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. 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.ProjectX.ProgramY.MyUnit1;
uses MyCompany.Libs.Unit2, Unit3, Unit4;
```

These declarations establish MyUnit1 as a member of the MyCompany.ProjectX.ProgramY 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 the current unit's namespace.

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 demonstrate the namespace search order:

```
// Project file declarations...
program MyCompany.ProjectX.ProgramY;

// Unit source file declaration...
unit MyCompany.ProjectX.ProgramY.MyUnit1;
```

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

1. MyCompany.ProjectX.ProgramY
2. MyCompany.ProjectX
3. 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!
```

Multi-unit Namespaces

Multiple units can belong to the same namespace, if the unit declarations refer to the same namespace. For example, one can create two files, unit1.pas and unit2.pas, with the following unit declarations:

```
// in file 'unit1.pas'
unit MyCompany.ProjectX.ProgramY.Unit1

// in file 'unit2.pas'
unit MyCompany.ProjectX.ProgramY.Unit2
```

In this example, the namespace MyCompany.ProjectX.ProgramY logically contains all of the **interface** symbols from unit1.pas and unit2.pas.

Symbol names in a namespace must be unique, across all units in the namespace. In the example above, it is an error for Unit1 and Unit2 to both define a global interface symbol named mySymbol.

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 just the name of the 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 fully-qualified name (including namespace and unit name) for the qualifier. The two forms of reference are identical and refer to the same symbol.

Note: Explicitly using a 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.

Fundamental Syntactic Elements Index

This section describes the fundamental syntactic elements, or the building blocks of the Delphi language.

Topics

- Fundamental Syntactic Elements (Delphi)
- Expressions (Delphi)
- Declarations and Statements (Delphi)

Fundamental Syntactic Elements (Delphi)

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.

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 encoding for its character set, including alphabetic and alphanumeric Unicode characters and the underscore. Delphi is not case-sensitive. The space character and control characters (U+0000 through U+001F including U+000D, the return or end-of-line character) are blanks.

The RAD Studio compiler will accept a file encoded in UCS-2 or UCS-4 if the file contains a byte order mark. The speed of compilation may be penalized by the use for formats other than UTF-8, however. All characters in a UCS-4 encoded source file must be representable in UCS-2 without surrogate pairs. UCS-2 encodings with surrogate pairs (including GB18030) are accepted only if the codepage compiler option is specified.

Tokens

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 in two lines, 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 pairs of symbols used in Delphi that have similar meanings (the symbol pairs {} and (* *) are comment characters that are further described in Comments and Compiler Directives):

Special Symbols	Similar Special Symbols
[ ]	(. .)
{ }	(* *)

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

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

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

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

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

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, a Unicode character, or an underscore (_) and cannot contain spaces. Alphanumeric characters, Unicode characters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers. 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 section Unit References and the Uses Clause in Programs and Units (Delphi).

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 do not qualify an identifier, its interpretation is determined by the rules of scope described in Blocks and scope inside Declarations and Statements (Delphi).

Extended Identifiers

You might encounter identifiers (e.g. types, or methods in a class) having the same name as a Delphi language reserved word. For example, a class might have a method called begin. Delphi reserved words such as **begin** cannot be used for an identifier name.

If you fully qualify the identifier, then there is no problem. For example, if you want to use the Delphi reserved word **type** for an identifer name, you must use its fully qualified name:

```
 var TMyType.type
 // Using a fully qualified name avoids ambiguity with {{Delphi}} language
keyword.
```

As a shorter alternative, the ampersand (**&**) operator can be used to resolve ambiguities between identifiers and Delphi language reserved words. The **&** prevents a keyword from being parsed as a keyword (that is, a reserved word). 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. But when you are **declaring** an identifier that has the same name as a keyword, you *must* use the **&**:

```
type
 &Type = Integer;
 // Prefix with '&' is ok.
```

Reserved Words

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

Delphi Reserved Words:

and	end	interface	record	var
array	except	is	repeat	while
as	exports	label	resourcestring	with
asm	file	library[3]	set	xor
begin	finalization	mod	shl	
case	finally	nil	shr	
class	for	not	string	
const	function	object	then	
constructor	goto	of	threadvar	
destructor	if	or	to	
dispinterface	implementation	packed	try	
div	in	procedure	type	
do	inherited	program	unit	
downto	initialization	property	until	
else	inline	raise	uses	

Note: *In addition to the words in the preceding table,* ***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. The keywords* ***of object*** *are used to define method pointers.*

Directives

Delphi has more than one type of directive. One meaning for "directive" is a word that is sensitive in specific locations within source code. This type of directive has special meaning in the Delphi language, but, unlike a reserved word, appears 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	export[12]	name	public	stdcall
abstract	external	near[1]	published	strict
assembler[12]	far[1]	nodefault	read	stored
automated	final	operator[10]	readonly	unsafe
cdecl	forward	out	reference[9]	varargs
contains[7]	helper[8]	overload	register	virtual
default	implements	override	reintroduce	winapi[6]
delayed	index	package[7]	requires[7]	write
deprecated[11]	inline[2]	pascal	resident[1]	writeonly
dispid	library[3]11	platform[11]	safecall	
dynamic	local[4]	private	sealed[5]	
experimental[11]	message	protected	static	

Note:

1. ***far***, ***near***, *and* ***resident*** *are obsolete.*
2. ***inline*** *is used directive-style at the end of procedure and function declaration to mark the procedure or function for inlining , but became a reserved word for Turbo Pascal.*
3. ***library*** *is also a keyword when used as the first token in project source code; it indicates a DLL target. Otherwise, it marks a symbol so that it produces a library warning when used.*
4. ***local*** *was a Kylix directive and is ignored for Delphi for Win32.*
5. ***sealed*** *is a class directive with odd syntax: 'class sealed'. A sealed class cannot be extended or derived (like* `final` *in C++).*
6. ***winapi*** *defines the default platform calling convention. For example, on Win32* ***winapi*** *is the same as* ***stdcall****.*
7. ***package****, when used as the first token, indicates a package target and enables package syntax.* ***requires*** *and* ***contains*** *are directives only in package syntax.*
8. ***helper*** *indicates "class helper for".*

9. ***reference*** *indicates a reference to a function or procedure.*
10. ***operator*** *indicates class operator.*
11. ***platform***, ***deprecated***, ***experimental***, *and* ***library*** *are hinting (or warning) directives. These directives produce warnings at compile time.*
12. ***assembler*** *and* ***export*** *directives have no meaning. They exist only for the backward compatibility.*

Types of Directives

Delphi has two types of directives, including the context-sensitive type of directive listed in the Directives table above.

A context-sensitive **directive** can be an identifier -- not typically a keyword -- that you place at the end of a declaration to modify the meaning of the declaration. For example:

```
procedure P; forward;
```

Or:

```
procedure M; virtual; override;
```

Or:

```
property Foo: Integer read FFoo write FFoo default 42;
```

The last type of directive is the official **compiler directive**, which is a switch or option that affects the behavior of the compiler. A compiler directive is surrounded by braces ({}) and begins with a dollar-sign ($), like this:

```
{$POINTERMATH ON}

{$D+} // DEBUGINFO ON
```

Like the other types of directives, **compiler directives** are not keywords. For a list of the compiler directives, see the **Delphi compiler directives list**.

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 preceding **-** 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 two's complement integer representation.

For more information about real and integer types, see About Data Types (Delphi). For information about the data types of numerals, see Declared Constants.

Labels

You can use either an identifier or a non-negative integer number as a label. The Delphi compiler allows numeric labels from 0 to 4294967295 (uint32 range).

Labels are used in **goto** statements. For more information about **goto** statements and labels, see Goto Statements in Declarations and Statements (Delphi).

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 characters, from an ANSI or multibyte 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.

The string is represented internally as a Unicode string encoded as UTF-16. Characters in the Basic Multilingual Plane (BMP) take 2 bytes, and characters not in the BMP require 4 bytes.

For example:

```
'Embarcadero'        { Embarcadero }
'You''ll see'        { You'll see }
'アプリケーションを Unicode 対応にする'
''''                 { ' }
''                   { 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 65,535 (decimal) or from $0 to $FFFF (hexadecimal) in UTF-16 encoding, and denotes the character corresponding to a specified code value. Each integer is represented internally by 2 bytes in the string. This is useful for representing control characters and multibyte characters. 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-return line-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 is compatible with any string type and with the **PChar** type. Since an **AnsiString** type may contain multibyte characters, a character string with one character, single or multibyte, is compatible with any character type. 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 About Data Types (Delphi).

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 left and right braces is a comment. }
(* Text between left-parenthesis-plus-asterisk and an
 asterisk-plus-right-parenthesis is also a comment *)
// Text between double-slash and end of line is a comment.
```

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

Here are some recommendations about how and when to use the three types of comment characters:

- Use the double-slash (//) for commenting out temporary changes made during development. You can use the Code Editor convenient `CTRL+/` (slash) mechanism to quickly insert the double-slash comment character while you are working.
- Use the parenthesis-star "(*...*)" both for development comments and for commenting out a block of code that contains other comments. This comment character permits multiple lines of source, including other types of comments, to be removed from consideration by the compiler.
- Use the braces ({}) for in-source documentation that you intend to remain with the code.

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.

in

`in` is a Delphi reserved word that may be used to:

- Check if an item exists in a set
- Iterate over the items of a container

Expressions (Delphi)

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 About Data

Types (Delphi)). 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 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; for details, see Variant Types (Delphi).

The sections that follow assume some familiarity with Delphi data types; for more information, see About Data Types (Delphi).

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 run-time 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 will not 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 run-time 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 run-time 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 shl 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.

Example

If `x` is a negative integer, the `shl` and `shr` operations are made clear in the following example:

```
var
  x: integer;
  y: string;

...
begin
  x := -20;
  x := x shr 1;
  //As the number is shifted to the right by 1 bit, the sign bit's value
replaced is with 0 (all negative numbers have the sign bit set to 1).
  y := IntToHex(x, 8);
  writeln(y);
  //Therefore, x is positive.
  //Decimal value: 2147483638
  //Hexadecimal value: 7FFFFFF6
  //Binary value: 0111 1111 1111 1111 1111 1111 1111 0110
end.
```

String Operators

The relational operators **=**, **<>**, **<**, **>**, **<=**, and **>=** all take string operands (see Relational operators later in this section). 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 (**UnicodeString**, **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 **PAnsiChar** and **PWideChar** (see Relational operators). The following operators also take pointers as operands. For more information about pointers, see Pointers and Pointer Types (Delphi) in About Data Types (Delphi).

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 **PAnsiChar** pointers; for **PWideChar** pointers `P + I` adds `I * 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 **PAnsiChar** (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 (Delphi).

Class and Interface 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 (Delphi), Object Interfaces (Delphi) and Interface References (Delphi).

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" in About Data Types (Delphi). 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" in About Data Types (Delphi).) 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 (Delphi).

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

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 have 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 (Delphi).

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 Structured Types (Delphi) in About Data Types (Delphi).

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 Data Types, Variables and Constants. For information about arrays and array properties, see Arrays in Data Types, Variables, and Constants and "Array Properties" in Properties (Delphi) page.

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;
  PByte = ^Byte;

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 (Delphi). For information about casting class and interface types, see "The as Operator" in Class References and Interface References (Delphi).

Declarations and Statements (Delphi)

This topic describes the syntax of Delphi declarations and statements.

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

This topic covers the following items:

- Declarations
- Simple statements such as assignment
- Structured statements such as conditional tests (for example, if-then, and **case**), iteration (for example, 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, interface, 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 produces 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), **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 compiler also recognizes the hinting directive **experimental**. You can use this directive to designate units that are in an unstable development state. The compiler will emit a warning when it builds an application that uses the unit.

For more information about the Delphi hinting directives, see **warning directives in method declarations**. All the Delphi directives are listed in Directives.

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 does not 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, the variable can

be replaced with the name of the function being defined. See Procedures and Functions (Delphi).) The **:=** symbol is sometimes called the **assignment operator**.

An assignment statement replaces the current value of the variable with the value of the 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 such as calls to procedures can be treated as statements in their own right:

```
MyFunction(X);
```

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

For more information about procedures and functions, see Procedures and Functions (Delphi).

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, you must 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 from 0 through 4294967295.

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, because 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 (Delphi).

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 this could have been written 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 code using a **with** statement:

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

or you could write the following code without using a **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*.

Since a **with** statement requires a variable or a field to operate upon, using it with properties can be tricky at times. A **with** statement expects variables it operates on to be available **by reference**.

The most important things to note when you are using **with**:

- You can use **with** on read-only properties *only for reading*. Trying to modify a field in the exposed record or object results in a compile-time error.
- Even though the property allows write access to the field, you still cannot use **with** to modify its fields.

The following code exemplifies the problem in using the **with** statement on read-only properties exposing a record. Assuming you have the following class:

```
TShape = class
private
  FCenter: TPoint;
public
  property Center: TPoint read FCenter;
end;
```

where TPoint is a records declared as follows:

```
TPoint = record
  X, Y: Integer;
end;
```

Normally, the **Center** property is read-only and does not allow you to modify the value or the fields of **FCenter** field. In this case, using a **with** statement like the following will fail with a compile-time error since Shape.Center is not a variable and you cannot have a reference to it:

```
with Shape.Center do
begin
  X := 100;
  Y := 100;
end;
```

The tricky part when using the **with** statement comes for read/write properties. We have changed the original **TShape** class to allow write access to its **FCenter** field:

```
TShape = class
private
  FCenter: TPoint;
public
  property Center: TPoint read FCenter '''write FCenter''';
end;
```

Even though the **Center** property is not read-only, the same compile-time error is emitted. The solution to this problem is to change code that looks like this:

```
with Shape.Center do
begin
  X := 100;
  Y := 100;
end;
```

into code that looks like this:

```
{ Copy the value of Center to a local variable. }
TempPoint := Shape.Center;

with TempPoint do
begin
  X := 100;
  Y := 100;
end;

{ Set the value back. }
Shape.Center := TempPoint;
```

If Statements

There are two forms of the **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 a semicolon between the **then** clause and the word **else** is never used. 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. This happens 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 **if**s. Consider a statement of the form:

```
if expression1 then if expression2 then statement1 else statement2;
```

It appears that there are two ways to parse this:

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

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

However, 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 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 smaller than 32 bits (string types and ordinals larger than 32 bits 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 Declared Constants.)
- 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 loops: **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.

Warning: The iteration variable *counter* cannot be modified within the loop. This includes assignment and passing the variable to a **var** parameter of a procedure. Doing so results in a compile-time warning.

For purposes of controlling the 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 the 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

Delphi supports 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;
- for Element in Record 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.

> **Warning:** 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 results in a compile-time warning.

Array expressions can 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 following code shows an example of traversing single, multidimensional, 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;
  DynArray: 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 multidimensional 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 }
  DynArray := TGenericIntArray.Create(1, 2, 3, 4);

  for I in DynArray do
  begin
    { Do something with I... }
  end;
end.
```

The following 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 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;
  C: Char;

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 or interface, the class or interface must implement a prescribed collection pattern. A type that implements the collection pattern must have the following attributes:

- The class or interface 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 **for-in** loop calls this method first to ensure that the container is not empty.
- 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.

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

```
type
  TMyIntArray   = array of Integer;
  TMyContainerEnumerator = class;

  TMyContainer  = class
  public
    Values: TMyIntArray;
    function GetEnumerator: TMyContainerEnumerator;
  end;

  TMyContainerEnumerator = class
    Container : TMyContainer;
    Index     : Integer;
  public
    constructor Create(AContainer : TMyContainer);
    function GetCurrent: Integer;
    function MoveNext:   Boolean;
    property Current:    Integer read GetCurrent;
  end;

constructor TMyContainerEnumerator.Create(AContainer : TMyContainer);
begin
  inherited Create;
  Container := AContainer;
  Index     := - 1;
end;

function TMyContainerEnumerator.MoveNext: Boolean;
begin
  Result := Index < High(Container.Values);
  if Result then
    Inc(Index);
end;

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

function TMyContainer.GetEnumerator: TMyContainerEnumerator;
begin
  Result := TMyContainerEnumerator.Create(Self);
end;

var
  MyContainer : TMyContainer;
  I           : Integer;
  Counter     : Integer;
begin
  MyContainer        := TMyContainer.Create;
  MyContainer.Values := TMyIntArray.Create(100, 200, 300);

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

  Writeln('Counter = ', Counter, ' (should be 600)');
  ReadLn;
end.
```

Iteration Over Datasets Using For Statements

Delphi supports for-in syntax construction to iterate over datasets. The compiler recognizes the following dataset iteration pattern:

- for Record in Dataset do Smth;

where `Record` is represented by the TDataSet API. It is safe to assume, that Record is equal to Dataset.

The following code snippet iterates over a dataset in Delphi. This sample code explains how to output the values of the Name column to a Memo control.

```
var
 ds: TDataSet;
//
FQuery1.SQL.Text := 'SELECT Name FROM Table1';
Memo1.Lines.Clear;
for ds in FDQuery1 do
  Memo1.Lines.Add(ds.FieldByName('Name').AsString);
```

> **Note** The dataset enumeration is not a reenterable operation. This means that for a dataset you can use only one enumeration at each moment. If you need to simultaneously execute several for-in loops for the same dataset, use the TDataSet.View method instead (see later in this topic). In this scenario, in the for-in loop, a Record may be not equal to a Dataset.

The following code snippet illustrates how to use the TDataSet.View method to enumerate a dataset.

```
var
  ds: TDataSet;
//...
Memo1.Lines.Clear;
for ds in FDQuery1.View(dmAllowClone) do
  Memo1.Lines.Add(ds.FieldByName('name').AsString);
```

List of Supported Classes

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

- System.Classes.TList
- System.Classes.TCollection
- System.Classes.TStrings
- System.Classes.TInterfaceList
- System.Classes.TComponent

- Vcl.Menus.TMenuItem
- Vcl.ActnList.TCustomActionList
- Vcl.ComCtrls.TListItems
- Vcl.ComCtrls.TTreeNodes
- Vcl.ComCtrls.TToolBar
- Data.DB.TFields
- Data.DB.TDataSet

Blocks and Scope

Declarations and statements are organized into *blocks* that 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 (Delphi).)

For example, in a function declaration like this:

```
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 (Delphi).)
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" in Structured Types (Delphi).)
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 (Delphi).)

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 does not 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" in Fundamental Syntactic Elements (Delphi)) or a **with** statement (see "With Statements" above.)

Data Types, Variables, and Constants Index

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

Topics

- About Data Types (Delphi)
- Simple Types (Delphi)
- String Types (Delphi)
- Structured Types (Delphi)
- Pointers and Pointer Types (Delphi)
- Procedural Types (Delphi)
- Variant Types (Delphi)
- Type Compatibility and Identity (Delphi)
- Data Types, Variables, and Constants Index (Delphi)
- Variables (Delphi)
- Declared Constants

About Data Types (Delphi)

This topic presents a high-level overview of Delphi data 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 run-time 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 **general**. 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 general 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 general. It is a good idea to use general types when possible, since they provide optimal performance and portability. However, changes in storage format from one implementation of a general 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**).
- Types can be **parameterized**, or **generic**, as well. Types can be generic in that they are the basis of a structure or procedure that operates in concert with different types determined later. For more information about generics or parameterized types, see the Generics Index.

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:

- In 32-bit platforms and 64-bit Windows `SizeOf(LongInt)` returns 4, since a **LongInt** variable uses four bytes of memory.
- In 64-bit iOS `SizeOf(LongInt)` returns 8, since a **LongInt** variable uses eight bytes of memory.

Type declarations are illustrated in the topics that follow. For general information about type declarations, see Data Types, Variables, and Constants Index (Delphi).

Simple Types (Delphi)

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

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 integral numbers.

Integer types can be *platform-dependent* and *platform-independent*.

Platform-Dependent Integer Types

The platform-dependent integer types are transformed to fit the bit size of the current compiler platform. The platform-dependent integer types are NativeInt, NativeUInt, LongInt, and LongWord. Using these types whenever possible, since they result in the best performance for the underlying CPU and operating system, is desirable. The following table illustrates their ranges and storage formats for the Delphi compiler.

Platform-dependent integer types

Type	Platform	Range	Format	Alias
NativeInt	32-bit platforms	-2147483648..2147483647 ($-2^{31}..2^{31}-1$)	Signed 32-bit	Integer
	64-bit platforms	-9223372036854775808..9223372036854775807 ($-2^{63}..2^{63}-1$)	Signed 64-bit	Int64
NativeUInt	32-bit platforms	0..4294967295 ($0..2^{32}-1$)	Unsigned 32-bit	Cardinal
	64-bit platforms	0..18446744073709551615 ($0..2^{64}-1$)	Unsigned 64-bit	UInt64

LongInt	32-bit platforms and 64-bit Windows platforms	-2147483648..2147483647 ($-2^{31}..2^{31}-1$)	Signed 32-bit	Integer
	64-bit POSIX platforms include iOS and Linux	-9223372036854775808..9223372036854775807 ($-2^{63}..2^{63}-1$)	Signed 64-bit	Int64
LongWord	32-bit platforms and 64-bit Windows platforms	0..4294967295 ($0..2^{32}-1$)	Unsigned 32-bit	Cardinal
	64-bit POSIX platforms include iOS and Linux	0..18446744073709551615 ($0..2^{64}-1$)	Unsigned 64-bit	UInt64

Note: 32-bit platforms include 32-bit Windows, 32-bit macOS, 32-bit iOS, iOS Simulator and Android.

Platform-Independent Integer Types

Platform-independent integer types always have the same size, regardless of what platform you use. Platform-independent integer types include ShortInt, SmallInt, LongInt, Integer, Int64, Byte, Word, LongWord, Cardinal, and UInt64.

Platform-independent integer types

Type	Range	Format	Alias
ShortInt	-128..127	Signed 8-bit	Int8
SmallInt	-32768..32767	Signed 16-bit	Int16
FixedInt	-2147483648..2147483647	Signed 32-bit	Int32
Integer	-2147483648..2147483647	Signed 32-bit	Int32
Int64	-9223372036854775808..9223372036854775807 ($-2^{63}..2^{63}-1$)	Signed 64-bit	
Byte	0..255	Unsigned 8-bit	UInt8
Word	0..65535	Unsigned 16-bit	UInt16
FixedUInt	0..4294967295	Unsigned 32-bit	UInt32
Cardinal	0..4294967295	Unsigned 32-bit	UInt32
UInt64	0..18446744073709551615 ($0..2^{64}-1$)	Unsigned 64-bit	

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 operands. Therefore, 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 character types are Char, AnsiChar, WideChar, UCS2Char, and UCS4Char:

- Char in the current implementation is equivalent to WideChar, since now the default string type is UnicodeString. Because the implementation of Char can change in future releases, it is 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.
- AnsiChar values are byte-sized (8-bit) characters ordered according to the locale character set, which is possibly multibyte.
- 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.
- UCS2Char is an alias for WideChar.
- UCS4Char is used for working with 4–byte Unicode characters.

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 WideChar; for example, **Chr(65)** returns the letter A.

AnsiChar and WideChar 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: AnsiChar;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do Inc(Letter);
end;
```

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

> **Note:** The AnsiChar type is not supported by the Delphi mobile compilers, but is used by the Delphi desktop compilers. For more information, see Migrating Delphi Code to Mobile from Desktop.

Boolean Types

The 4 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 2 bytes (one word), and a LongBool variable occupies 4 bytes (2 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 an expression that returns a Boolean value }
 ...
var OK: Boolean;        { use a Boolean variable }
  ...
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 on.

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 are writing an application and you create an event handler like:

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

you will get a compilation error; the compiler interprets **Click** within the scope of the procedure as a reference to a **Click** method of a TForm. 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 cannot 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;
begin
  X := Small;   // Ord(X) = 5
  X := Size(6); // Ord(X) = 6
  Inc(X);       // Ord(X) = 7
```

Any value that is not explicitly assigned an ordinality has the ordinality one greater than that of the previous value in the list. If the first value is not 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);
```

Scoped Enumerations

You can use scoped enumerations in Delphi code if you enable the {$SCOPEDENUMS ON} compiler directive.

The {$SCOPEDENUMS ON or OFF} compiler directive enables or disables the use of scoped enumerations in Delphi code. **{$SCOPEDENUMS ON}** defines that enumerations are scoped. **{$SCOPEDENUMS ON}** affects declarations of enumeration types until the nearest **{$SCOPEDENUMS OFF}** directive. The identifiers of the enumeration introduced in enumeration types declared after

the {$SCOPEDENUMS ON} directive are not added to the global scope. To use a scoped enumeration identifier, you should qualify it with the name of the enumeration type introducing this identifier.

For instance, let us define the following unit in the Unit1.pas file

```
unit Unit1;
interface
// {$SCOPEDENUMS ON} // clear comment from this directive
 type
   TMyEnum = (First, Second, Third);
implementation

end.
```

and the following program using this unit

```
program Project1;
{$APPTYPE CONSOLE}

uses
  SysUtils, Unit1 in 'Unit1.pas';

var
  // First: Integer;  // clear comment from this variable
  Value: TMyEnum;
begin
  try
    Value := First;
//  Value := TMyEnum.First;
//  Value := unit1.First;
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
end.
```

Now we can investigate effects of the {$SCOPEDENUMS} compiler directive on the scopes in which the `First`, `Second`, and `Third` identifiers, defined in the `TMyEnum` enumeration, are visible.

First, **Run (F9)** on this code. The code runs successfully. This means that the `First` identifier, used in the

```
Value := First;
```

variable, is the global scope identifier introduced in the

```
TMyEnum = (First, Second, Third);
```

enumeration type.

Now clear comment from the

```
{$SCOPEDENUMS ON}
```

compiler directive in the `unit1` unit. This directive enforces the `TMyEnum` enumeration to be scoped. Execute **Run**. The E2003 Undeclared identifier 'First' error is generated on the

```
Value := First;
```

line. It informs that the `{$SCOPEDENUMS ON}` compiler directive prevents the `First` identifier, introduced in the scoped `TMyEnum` enumeration, to be added to the global scope.

To use identifiers introduced in scoped enumerations, prefix a reference to an enumeration's element with its type name. For example, clear comment in the second

```
Value := TMyEnum.First;
```

version of the `Value` variable (and comment the first version of `Value`). Execute **Run**. The program runs successfully. This means that the `First` identifier is known in the `TMyEnum` scope.

Now comment the

```
// {$SCOPEDENUMS ON}
```

compiler directive in `unit1`. Then clear comment from the declaration of the `First` variable

```
First: Integer;
```

and again use the

```
Value := First;
```

variable. Now the code in the `program Project1` looks like this:

```
var
  First: Integer;
  Value: TMyEnum;
begin
  try
    Value := First;
```

Execute **Run**. The

```
 First: Integer;
```

line causes the E2010 Incompatible types - 'TMyEnum' and 'Integer' error. This means that the naming conflict occurs between the global scope `First` identifier introduced in the `TMyEnum` enumeration and the `First` variable. You can work around this conflict by qualifying the `First` identifier with the `unit1` unit in which it is defined. For this, comment again the first version of `Value` variable and clear comment from the third one:

```
Value := unit1.First;
```

Execute **Run**. The program runs successfully. That is, now the `First` identifier can be qualified with the `unit1` unit scope. But what happens if we again enable the

```
{$SCOPEDENUMS ON}
```

compiler directive in `unit1`. The compiler generates the E2003 Undeclared identifier 'First' error on the

```
Value := unit1.First;
```

line. This means that **{$SCOPEDENUMS ON}** prevents adding the `First` enumeration's identifier in the `unit1` scope. Now the `First` identifier is added only in the `TMyEnum` enumeration's scope. To check this, let us again use the

```
Value := TMyEnum.First;
```

version of the `Value` variable. Execute **Run** and the code succeeds.

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 through 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, the following code:

```
...
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 the floating-point notation. The table below gives the ranges and storage formats for the real types on 64-bit and 32-bit platforms.

Real types

Type	Platform	Approximate Positive Range	Significant decimal digits	Size in bytes
Real48	all	2.94e-39 .. 1.70e+38	11-12	6
Single	all	1.18e-38 .. 3.40e+38	7-8	4
Double	all	2.23e-308 .. 1.79e+308	15-16	8
Real	all	2.23e-308 .. 1.79e+308	15-16	8
Extended	32bit Intel Windows	3.37e-4932 .. 1.18e+4932	10-20	10
	64-bit Intel Linux 32-bit Intel macOS 32-bit Intel iOS Simulator	3.37e-4932 .. 1.18e+4932	10-20	16
	other platforms	2.23e-308 .. 1.79e+308	15-16	8
Comp	all	-9223372036854775807.. 9223372036854775807 $(-2^{63}+1..\ 2^{63}-1)$	10-20	8
Currency	all	-922337203685477.5807.. 922337203685477.5807	10-20	8

The following remarks apply to real types:

- Real is equivalent to Double, in the current implementation.

- 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 6-byte Real48 type was called Real in earlier versions of Object Pascal. If you are recompiling code that uses the older, 6-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 6-byte type.

- Extended offers greater precision on **32**-bit platforms than other real types.

On **64**-bit platforms Extended is an alias for a Double; that is, the size of the Extended data type is 8 bytes. Thus you have less precision using an Extended on **64**-bit platforms compared to **32**-bit platforms, where Extended is 10 bytes. Therefore, if your applications use the Extended data type and you rely on precision for floating-point operations, this size difference might affect your data. Be careful using Extended if you are creating data files to share across platforms. For more information, see The Extended Data Type Is 2 Bytes Smaller on 64-bit Windows Systems.

- 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. It is stored as a scaled 64-bit integer with the 4 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 (Delphi)

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

- Short strings (ShortString)
- ANSI strings (AnsiString)
- Unicode strings (UnicodeString and WideString)

All the string types described in this topic are supported by Delphi compilers for desktop platforms, but Delphi compilers for mobile platforms only support UTF8String, RawByteString and the default string type (UnicodeString). Also, with Delphi compilers for mobile platforms, strings are 0-based and immutable; to manipulate strings, use the TStringHelper functions, which are provided for this

purpose. For more information, see **Migrating Delphi Code to Mobile from Desktop**.

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, Unicode characters, etc.
UnicodeString **Note:** In RAD Studio, `string` is an alias for UnicodeString.	~2^30 characters	4 bytes to 2GB	Unicode characters, 8-bit (ANSI) characters, multiuser servers and multilanguage applications UnicodeString is the default string type.
WideString	~2^30 characters	4 bytes to 2GB	Unicode characters; multiuser servers and multilanguage applications. WideString is not supported by the Delphi compilers for mobile platforms, but is supported by the Delphi compilers for desktop platforms. Using UnicodeString is preferred to WideString.

Note: The default string type is UnicodeString. WideString is provided to be compatible with the COM `BSTR` type. You should generally use UnicodeString for non-COM applications. For most purposes UnicodeString is the preferred type. The type string is an alias for UnicodeString.

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 the `var` and `out` parameters) must be of the appropriate type. Strings can be explicitly cast to a different string type. However, casting a multibyte string to a single byte string may result in data loss.

There are some special string types worth mentioning:

- Code paged AnsiStrings are defined like this:

```
Type mystring = type AnsiString(CODEPAGE)
```

It is an AnsiString that has an affinity to maintaining its internal data in a specific code page.

- The RawByteString type is `type AnsiString($FFFF)`. RawByteString enables the passing of string data of any code page without doing any code page conversions. RawByteString should only be used as a `const` or value type parameter or a return type from a function. It should never be passed by reference (passed by `var`), and should never be instantiated as a variable.
- UTF8String represents a string encoded using UTF-8 (variable number of bytes Unicode). It is a code paged AnsiString type with a UTF-8 code page.

The reserved word string functions like a general string 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 UnicodeString.

On the Win32 platform, you can use the `{$H-}` directive to turn string into ShortString. This is a potentially useful technique when using older 16-bit Delphi code or Turbo Pascal code with your current programs.

Note that the keyword string is also used when declaring ShortString types of specific lengths (see **Short Strings**, below).

Comparison of strings is defined by the ordering of the elements 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 represent the lowest values.

You can index a string variable just as you would an array. If `S` is a non-UnicodeString string variable and `i`, an integer expression, `S[i]` represents the *ith* byte in `S`, which may not be the *ith* character or an entire character at all for a multibyte character string (MBCS). Similarly, indexing a UnicodeString variable results in an element that may not be an entire character. If the string contains characters in the Basic Multilingual Plane (BMP), all characters are 2 bytes, so indexing the string gets characters. However, if some characters are not in the BMP, an indexed element may be a surrogate pair - not an entire character.

The standard function `Length` returns the number of elements in a string. As noted above, the number of elements is not necessarily the number of characters. The `SetLength` procedure adjusts the length of a string. Note that the `SizeOf` function returns the number of bytes used to represent a variable or type. Note that `SizeOf` returns the number of characters in a string *only for a short string*. `SizeOf`

returns the number of bytes in a pointer for all other string types, since they are pointers.

For a short string 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 `UpCase` function to convert `MyString` to uppercase:

```
var I: Integer;
begin
  I := Length(MyString);
  while I > 0 do
  begin
    MyString[I] := UpCase(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 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 single-byte 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.

AnsiString

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

An AnsiString variable is a structure containing string information. When the variable is empty - that is, when it contains a zero-length string, the pointer is **nil** and the string uses no additional storage. When the variable is nonempty, it points to a dynamically allocated block of memory that contains the string value. This memory is allocated on the heap, but its management is entirely automatic and requires no user code. The AnsiString structure contains a 32-bit length indicator, a 32-bit reference count, a 16-bit data length indicating the number of bytes per character, and a 16-bit code page.

An AnsiString represents a single byte string. With a single-byte character set (SBCS), each byte in a string represents one character. In a multibyte character set (MBCS), the elements are still single bytes, but some characters are represented by one byte and others by more than one byte. Multibyte character sets - especially double-byte character sets (DBCS) - are widely used for Asian languages. An AnsiString can contain MBCS characters.

Indexing of AnsiString is 1-based. Indexing multibyte strings is not reliable, since `S[i]` represents the *ith* byte (not necessarily the *ith* character) in `S`. The *ith* byte may be a single character or part of a character. However, the standard AnsiString 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.

Because AnsiString variables have 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 an AnsiString variable is destroyed or assigned a new value, the reference count of the old AnsiString (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.

UnicodeString (the Default String Type)

The UnicodeString type is the default string type and represents a dynamically allocated Unicode string whose maximum length is limited only by available memory.

In a Unicode character set, each character is represented by one or more bytes. Unicode has several *Unicode Transformation Formats* that use different but equivalent character encodings that can be easily transformed into each other.

- In UTF-8, for instance, characters may be one to 4 bytes. In UTF-8, the first 128 Unicode characters map to the US-ASCII characters.
- UTF-16 is another commonly used Unicode encoding in which characters are either 2 bytes or 4 bytes. The majority of the world's characters are in the *Basic Multilingual Plane* and can be represented in 2 bytes. The remaining characters require two 2 byte characters known as *surrogate pairs*.
- UTF-32 represents each character with 4 bytes.

The Win32 platform supports single-byte and multibyte character sets as well as Unicode. The Windows operating system supports UTF-16.

See the Unicode Standard for more information.

The UnicodeString type has exactly the same structure as the AnsiString type. UnicodeString data is encoded in UTF-16.

Since UnicodeString and AnsiString have the same structure, they function very similarly. When a UnicodeString variable is empty, it uses no additional memory. When it is not empty, it points to a dynamically allocated block of memory that contains the string value, and the memory handling for this is transparent to the user. UnicodeString variables are reference counted, and two or more of them can reference the same value without consuming additional memory.

Instances of UnicodeString can index characters. Indexing is 1-based, just as for AnsiString.

UnicodeString is assignment compatible with all other string types. However, assignments between AnsiString and UnicodeString do the appropriate up or down conversions. Note that assigning a UnicodeString type to an AnsiString type is not recommended and can result in data loss.

Delphi can also support Unicode characters and strings through the WideChar, PWideChar, and WideString types.

For more information on using Unicode, see Unicode in RAD Studio and Enabling Applications for Unicode.

WideString

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

WideString is appropriate for use in COM applications. However, WideString is not reference counted, and so UnicodeString is more flexible and efficient in other types of applications.

Indexing of WideString multibyte strings is not reliable, since `S[i]` represents the *ith* element (not necessarily the *ith* character) in `S`.

For Delphi, Char and PChar types are WideChar and PWideChar types, respectively.

Note:

`WideString` is not supported by the Delphi compilers for mobile platforms, but is used by the Delphi compilers for desktop platforms.

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 Input-Output) 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 (Delphi).) 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 the original constant string '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 strings (AnsiString and UnicodeString values) and null-terminated strings (PChar values) in expressions and assignments, and you can pass PChar values to functions or procedures that take 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 string.

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

You can cast a PChar value as a UnicodeString. 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 UnicodeString or AnsiString string as a null-terminated string. The following rules apply:

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

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

Use `PAnsiChar(S)` if `S` is an AnsiString.

- You can also use `Pointer(S)` to cast a 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 UnicodeString or AnsiString 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 string expression to a pointer, the pointer is valid only within the statement where the typecast is performed.
- When you cast a UnicodeString or AnsiString expression to a pointer, the pointer should usually be considered read-only. You can safely use the pointer to modify the string only when all of the following conditions are satisfied:

The expression cast is a UnicodeString or AnsiString 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 procedures.

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 (Delphi)

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.

This topic covers the following structured types:

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

Alignment of Structured Types

By default, the values in a structured type are aligned on word- or double-word boundaries for faster access.

You can, however, specify byte alignment by including the reserved word **packed** when you declare a structured type. The **packed** word specifies compressed data storage. Here is an example declaration:

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

Using **packed** is not a recommended practice, because it can prevent compatibility with other languages or platforms, it slows data access, and, in the case of a character array, it affects type compatibility. For more information, see Memory management and Implicit Packing of Fields with a Common Type Specification.

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 set 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'];
```

Note: For more information, see the following warning message: **W1050 WideChar reduced to byte char in set expressions (Delphi)**.

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 []. For more information about sets, see "Set Constructors" and "Set Operators" in Expressions (Delphi).

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 (Delphi).

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" in String Types (Delphi).

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. An alternative method of allocating memory for dynamic arrays is to invoke the array constructor:

```
type
  TMyFlexibleArray = array of Integer;

begin
  MyFlexibleArray := TMyFlexibleArray.Create(1, 2, 3 {...});
end;
```

which allocates memory for three elements and assigns each element the given value.

Similar to the array constructor, a dynamic array may also be initialized from an array constant expression as below.

```
procedure MyProc;
var
  A: array of Integer;
begin
  A := [1, 2, 3];
end;
```

Notice that unlike with an array constructor, an array constant can be applied to unnamed dynamic array type directly. This syntax is specific to dynamic arrays; applying this technique to other array types is likely to result in the constant being interpreted as a set, leading to an incompatible types error at compile-time.

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, either of the following truncates all but the first 20 elements of `A`:

```
SetLength(A, 20)


A := Copy(A, 0, 20)
```

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;
  Int2: 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;
```

String-Like Operations Supported on Dynamic Arrays

Dynamic arrays can be manipulated similarly to strings. For example:

```
var
  A: array of integer;
  B: TBytes = [1,2,3,4]; //Initialization can be done from declaration
begin
  ...
  A:=[1,2,3]; // assignation using constant array
  A:=A+[4,5]; // addition - A will become [1,2,3,4,5]
  ...
end;
```

String-like Support Routines

Some of the Delphi Intrinsic Routines support operations on dynamic arrays in addition to operations on strings.

System.Insert The **Insert** function inserts a dynamic array at the beginning at the position index. It returns the modified array:

```
var
  A: array of integer;
begin
  ...
  A:=[1,2,3,4];
  Insert(5,A,2); // A will become [1,2,5,3,4]
  ...
end;
```

System.Delete

The **Delete** function eliminates elements from a dynamic array and returns the modified array:

```
var
  A: array of integer;
begin
  ...
  A:=[1,2,3,4];
  Delete(A,1,2); //A will become [1,4]
  ...
end;
```

System.Concat The **Concat** function can be used to put together two different dynamic arrays:

```
  A := Concat([1,2,3],[4,5,6]); //A will become [1,2,3,4,5,6]
```

Records (traditional)

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).

Records (advanced)

In addition to the traditional record types, the Delphi language allows more complex and "class-like" record types. In addition to fields, records may have properties and methods (including constructors), class properties, class methods, class fields, and nested types. For more information on these subjects, see the documentation on Classes and Objects (Delphi). Here is a sample record type definition with some "class-like" functionality.

```
type
  TMyRecord = record
    type
      TInnerColorType = Integer;
    var
      Red: Integer;
    class var
      Blue: Integer;
    procedure printRed();
    constructor Create(val: Integer);
    property RedProperty: TInnerColorType read Red write Red;
    class property BlueProp: TInnerColorType read Blue write Blue;
end;

constructor TMyRecord.Create(val: Integer);
begin
  Red := val;
end;

procedure TMyRecord.printRed;
begin
  Writeln('Red: ', Red);
end;
```

Though records can now share much of the functionality of classes, there are some important differences between classes and records.

- Records do not support inheritance.
- Records can contain variant parts; classes cannot.

- Records are value types, so they are copied on assignment, passed by value, and allocated on the stack unless they are declared globally or explicitly allocated using the New and Dispose function. Classes are reference types, so they are not copied on assignment, they are passed by reference, and they are allocated on the heap.
- Records allow operator overloading on the Win32 platform; classes, however, do not allow operator overloading.
- Records are constructed automatically, using a default no-argument constructor, but classes must be explicitly constructed. Because records have a default no-argument constructor, any user-defined record constructor must have one or more parameters.
- Record types cannot have destructors.
- Virtual methods (those specified with the **virtual**, **dynamic**, and **message** keywords) cannot be used in record types.
- Unlike classes, record types on the Win32 platform cannot implement interfaces.

File Types (Win32)

File types, as available on the Win32 platform, are sequences 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 Input-Output under the "File Input and Output" section.

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" in Standard Routines and Input-Output.

Files are not allowed in arrays or records.

Code Samples

- ComplexNumbers Sample (Records)

Pointers and Pointer Types (Delphi)

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. The PByte type is used for any byte data that is not character data.

On 32-bit platforms, a pointer occupies four bytes of memory as a 32-bit address. On 64-bit platforms, a pointer occupies eight bytes of memory as a 64-bit address.

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

```
^typeName
```

the caret symbol denotes a type that represents pointers to variables of type `typeName`.

When the caret symbol appears after a `pointer` variable:

```
pointer^
```

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

This example might 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.

Using Extended Syntax with Pointers

The **{$EXTENDED}** compiler directive affects the use of the caret (^). When {$X+} is in effect (the default), you can omit the caret when referencing pointers. The caret is still required when declaring a pointer and for resolving the ambiguity when a pointer points to another pointer. For more information, see Extended syntax (Delphi).

With extended syntax enabled, you can omit the caret when referring to a pointer, as in the following example:

```
{$X+}
 type
   PMyRec = ^TMyRec;
   TMyRec = record
     Data: Integer;
   end;

 var
   MyRec: PMyRec;

 begin
   New(MyRec);
   MyRec.Data := 42;  {#1}
 end.
```

When extended syntax is not enabled, the line marked {#1} would typically be expressed as:

```
 MyRec^.Data := 42;
```

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 might be useful to also define a pointer to that type. This makes it easy to manipulate instances of the type without copying large blocks of memory.

> **Note:** You can declare a pointer type before you declare the type it points to.

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 a WideChar). These character pointers are used to manipulate null-terminated strings. (See "Working with null-terminated strings" in String Types (Delphi).)

> **Note:** Do not cast non-character pointer types to PChar to do pointer arithmetic. Instead, use the PByte pointer type, which is declared with the {$POINTERMATH ON} compiler directive.

Byte Pointer

The fundamental type PByte represents a pointer to any byte data that is not character data. This type is declared with the `{$POINTERMATH ON}` compiler directive:

```
function TCustomVirtualStringTree.InternalData(Node: PVirtualNode): Pointer;
begin
  if (Node = FRoot) or (Node = nil) then
    Result := nil
  else
    Result := PByte(Node) + FInternalDataOffset;
end;
```

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.

Use the `{POINTERMATH <ON|OFF>}` directive to turn pointer arithmetic on or off for all typed pointers, so that increment/decrement is by element size.

Selected pointer types declared in System and SysUtils

Pointer type	Points to variables of type
PString	UnicodeString
PAnsiString	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 (Delphi)

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions.

This topic does not refer to the newer type of procedural type used with anonymous methods, that is, a "reference to a procedure". See Anonymous Methods in Delphi.

About Procedural Types

The following example demonstrates usage of a procedural type. 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 is left is the right part of a procedural type declaration. 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
```

Method Pointers

The variables shown in the previous example are all procedure pointers - that is, pointers to the address of a procedure or function. If you want to reference a method of an instance object (see Classes and Objects (Delphi)), 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 (Delphi)

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 run time. 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 run-time 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 (Delphi).) They can hold dynamic arrays, and they can hold a special kind of static array called a variant array. (See "Variant arrays" later in this chapter.) 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 run-time 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 System.Variants unit.

> **Note:** Variant records are considered inherently "unsafe." A variant record is very similar to using the "absolute" directive because the variant field parts of the record are literally overlaid in memory atop each other. You can assign a value as one type and then read it out as a different type. If you are using variants, you might see compiler warnings about unsafe code, such as W1047 Unsafe code '%s' (Delphi).

On 32-bit platforms, a variant is stored as a 16-byte record. On 64-bit platforms, a variant is stored as a 24-byte record. A variant record consists of a type code and a value, or a pointer to a value, of the type specified by the type 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: Source:	integer	real	string	Boolean
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**: all bits set to 1 (-1 if Integer, 255 if Byte, etc.)	**False** = 0, **True** = 1	**False** = 'False', **True** = 'True' by default; casing depends on global variable System.Variants.BooleanToStringRule.	**False** = **False**, **True** = **True**
Unassigned	Returns 0.	Returns 0.	Returns empty string.	Returns **False**.
Null	Depends on global variable System.Variants.NullStrictConvert (raises an exception by default).	Depends on global variable System.Variants.NullStrictConvert (raises an exception by default).	Depends on global variables System.Variants.NullStrictConvert and System.Variants.NullAsStringValue (raises an exception by default).	Depends on global variable System.Variants.NullStrictConvert (raises an exception by default).

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 Variants.EVariantError exception or an exception class descending from Variants.EVariantError.

Special conversion rules apply to the System.TDateTime type declared in the System unit. When a System.TDateTime is converted to any other type, it treated as a normal Double. When an integer, real, or Boolean is converted to a System.TDateTime, it is first converted to a Double, then read as a date-time value. When a string is converted to a System.TDateTime, it is interpreted as a date-time value using the regional settings. When an Unassigned value is converted to System.TDateTime, it is treated like the real or integer value 0. Converting a Null value to System.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. Do not perform such operations unnecessarily, since they are memory-inefficient.

OleVariant

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 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 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 (Delphi)

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 **PAnsiChar** or **PWideChar** and the other is a zero-based character array of the form array[0..n] of **PAnsiChar** 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 **PAnsiChar**, **PWideChar**, **PChar** 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 **PAnsiChar**, **PWideChar** or **PChar**.

- 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 **PAnsiChar** or **PWideChar** and T2 is a zero-based character array of the form array[0..n] of Char (when T1 is **PAnsiChar**) 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. See "Procedural types in statements and expression" earlier in this chapter.)
- 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.)

Data Types, Variables, and Constants Index (Delphi)

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 run time, 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 information, for 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 KnownType
```

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 (Delphi)

A variable is an identifier whose value can change at run time. 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;
```

Local variables cannot be initialized in their declarations. Multiple variable declarations (such as var X, Y, Z: Real;) cannot include initializations, nor can declarations of variant and file-type variables.

If you do not explicitly initialize a global variable, the compiler initializes it to 0. Object instance data (fields) are also initialized to 0. On the Wiin32 platform, the contents of a local variable 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 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 compiler (long strings, wide strings, dynamic arrays, variants, and interfaces) can 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	Aliases
0 $FF	0 255	**Byte**	**UInt8**
0 $FFFF	0 65535	**Word**	**UInt16**
0 $FFFFFFFF	0 4294967295	**Cardinal**	**UInt32**, **FixedUInt**
0 $FFFFFFFFFFFFFFFF	0 18446744073709551615	**UInt64**	
-$80 $7F	-128 127	ShortInt	**Int8**
-$8000 $7FFF	-32768 32767	**SmallInt**	**Int16**
-$80000000 $7FFFFFFF	-2147483648 2147483647	**Integer**	**Int32**, **FixedInt**
-$8000000000000000 $7FFFFFFFFFFFFFFF	-9223372036854775808 9223372036854775807	**Int64**	

32-bit native integer type

Range of constant (hexadecimal)	Range of constant (decimal)	Type	Equivalent type
-$80000000 $7FFFFFFF	-2147483648 2147483647	**NativeInt**	**Integer**
0 $FFFFFFFF	0 4294967295	**NativeUInt**	**Cardinal**

64-bit native integer type

Range of constant (hexadecimal)	Range of constant (decimal)	Type	Equivalent type
-$8000000000000000 $7FFFFFFFFFFFFFFF	-9223372036854775808 9223372036854775807	**NativeInt**	**Int64**
0 $FFFFFFFFFFFFFFFF	0 18446744073709551615	**NativeUInt**	**UInt64**

32-bit platforms and 64-bit Windows integer type

32-bit platforms include 32-bit Windows, OSX32, 32-bit iOS, and Android.

Range of constant (hexadecimal)	Range of constant (decimal)	Type	Equivalent type
-$80000000 $7FFFFFFF	-2147483648 2147483647	**LongInt**	**Integer**
0 $FFFFFFFF	0 4294967295	**LongWord**	**Cardinal**

64-bit platforms integer type

64-bit platforms include 64-bit iOS.

Range of constant (hexadecimal)	Range of constant (decimal)	Type	Equivalent type
-$8000000000000000 $7FFFFFFFFFFFFFFF	-9223372036854775808 9223372036854775807	**LongInt**	**Int64**
0 $FFFFFFFFFFFFFFFF	0 18446744073709551615	**LongWord**	**UInt64**

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)
'Embarcadero' + ' ' + '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 as 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 = 'Embarcadero 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. 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 elements of the array, 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 resolved at least as a relative address at 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!';
```

Writeable Typed Constants

Delphi allows typed constants to be modified if you set the compiler directive {$J+} or Writeable typed constants (Delphi) {$WRITEABLECONST ON}.

With $J+ set, you can use assignment statements to change the value of typed constants as if they were initialized variables. For example:

```
{$J+}
const
  foo: Integer = 12;
begin
  foo := 14;
end.
```

Differences between writeable typed constants and initialized variables:

- Writeable typed constants can occur both globally and locally in procedure, functions and methods.
- Initialized variables are only available as global declarations.
- Initialized variables cause a compile-time error when attempted within procedures or methods.

Procedures and Functions Index

This section describes the syntax of function and procedure declarations.

Topics

Procedures and Functions (Delphi)

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 body of the routine 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 statement block of a procedure, 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 do not 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 statement block of the function 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 return value of the function, 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 return value of the function. 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**, **safecall**, and **winapi**. 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**.

- For the **register** and **pascal** conventions, the evaluation order is not defined.
- 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.
- **winapi** is not actually a calling convention. **winapi** defines using the default platform calling convention. For example, on Win32 **winapi** is the same as **stdcall**.

The table below summarizes calling conventions.

Calling conventions :

Directive	Parameter order	Clean-up	Passes parameters in registers?
register	Undefined	Routine	Yes
pascal	Undefined	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 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 parameter list or return type of the routine, 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 cannot 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 32-bit Windows

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

imports a function called `SomeFunction` from `strlib.dll`.

Importing a Routine Under a Different Name

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 `stringConstant1` gives the name of the library file and `stringConstant2` is the original name of the routine.

The following declaration imports a function from `user32.dll` (part of the Windows API):

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer):
Integer; stdcall; external 'user32.dll' name 'MessageBoxA';
```

The original name of the function is `MessageBoxA`, but it is imported as `MessageBox`.

In your importing declaration, be sure to match the exact spelling and case of the name of the routine. Later, when you call the imported routine, the name is case-insensitive.

Importing a Routine by Index

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 index of the routine in the export table.

Delaying the Loading of the Library

To postpone the loading of the library that contains the function to the moment the function is actually needed, append the `delayed` directive to the imported function:

```
function ExternalMethod(const SomeString: PChar): Integer; stdcall; external
'cstyle.dll' delayed;
```

`delayed` ensures that the library that contains the imported function is not loaded at application startup, but rather when the first call to the function is made. For more information on this topic, see the Libraries and Packages - Delayed Loading topic.

Specifying Dependencies of the Library

If the library that contains the target routine depends on other libraries, use the `dependency` directive to specify those dependencies.

To use the `dependency` directive, append the `dependency` keyword followed by a comma-separated list of strings. Each string must contain the name of a library that is a dependency of the target external library:

```
external <library> dependency <dependency1>, <dependency2>, …
```

The following declaration indicates that `libmidas.a` depends on the standard C++ library:

```
function DllGetDataSnapClassObject(const [REF] CLSID, [REF] IID: TGUID; var
Obj): HResult; cdecl; external 'libmidas.a' dependency 'stdc++';
```

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 declarations of the routine, 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 parameter list of the routine.

The compiler can distinguish between overloaded functions that contain **AnsiString**/**PAnsiChar**, **UnicodeString**/**PChar** and **WideString**/**PWideChar** 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 **UnicodeString**/**PChar**.

```
procedure test(const A: AnsiString); overload;
procedure test(const W: WideString); overload;
procedure test(const U: UnicodeString); overload;
procedure test(const PW: PWideChar); overload;
var
  a: AnsiString;
  b: WideString;
  c: UnicodeString;
  d: PWideChar;
  e: string;
begin
  a := 'a';
  b := 'b';
  c := 'c';
  d := 'd';
  e := 'e';
  test(a);    // calls AnsiString version
  test(b);    // calls WideString version
  test(c);    // calls UnicodeString version
  test(d);    // calls PWideChar version
  test(e);    // calls UnicodeString version
  test('abc');    // calls UnicodeString version
  test(AnsiString ('abc'));    // calls AnsiString version
  test(WideString('abc'));    // calls WideString version
  test(PWideChar('PWideChar'));    // calls PWideChar 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 -> 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 name of a routine 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 statement block of the routine. 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 (Delphi)

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 - DoubleByRef can 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 variable to 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 cannot 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.

Constant parameters may be passed to the function by value or by reference, depending on the specific compiler used. To force the compiler to pass a constant parameter by reference, you can use the **[Ref]** decorator with the **const** keyword.

The following example shows how you can specify the **[Ref]** decorator either before or after the **const** keyword**:**

```
function FunctionName(const [Ref] parameter1: Class1Name; [Ref] const
parameter2: Class2Name);
```

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 does not 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. 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;  // Integer occupies 4 bytes. Therefore 8 bytes in a whole
  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, 8);                    // compare Vec1[1] to P.X and Vec1[2]
to P.Y
                                         // each Vec1[x] is integer and
occupies 4 bytes
```

String Parameters

When you declare routines that take short-string parameters, you cannot include length specifiers in the parameter declarations. That is, the following declaration causes a compilation error:

```
procedure Check(S: string[20]);   // syntax error
```

But the following declaration is valid:

```
type TString20 = string[20];
procedure Check(S: TString20);
```

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 Length - 1, 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`. System.TVarRec, which is 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, strings are passed as **Pointer** and must be typecast to **string**.

The following Win32 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);
          vtUnicodeString:  Result := Result + string(VUnicodeString);
          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 the following declaration is legal:

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

The following declaration is not legal:

```
function MyFunction(X, Y: Real = 3.5): Real;  // syntax error
```

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 values if there are any must 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 (Delphi)

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 declared name of the routine (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 parameter list of the routine. The parameters you pass to a routine are called actual parameters, while the parameters in the declaration of the routine 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 the formal parameters of a routine 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 are not defined before use 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 is done between units that are circularly dependent. This includes 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.
- 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.
- Within a unit, the body for an inline function should be defined before calls to the function are made. Otherwise, the body of the function, which is not known to the compiler when it reaches the call site, 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 definition of the routine, 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 on whether a routine will be inlined when it is used at the call site of the routine.
{$INLINE OFF}	The routine will not be marked as inlineable, even if it is tagged with inline.	The routine will not be expanded inline.

Anonymous Methods in Delphi

As the name suggests, an *anonymous method* is a procedure or function that does not have a name associated with it. An anonymous method treats a block of code as an entity that can be assigned to a variable or used as a parameter to a method. In addition, an anonymous method can refer to variables and bind values to the variables in the context in which the method is defined. Anonymous methods can be defined and used with simple syntax. They are similar to the construct of *closures* defined in other languages.

Note: This topic covers handling Delphi anonymous method in Delphi code. For C++ code, see **How to Handle Delphi Anonymous Methods in C++**.

Syntax

An anonymous method is defined similarly to a regular procedure or function, but with no name.

For example, this function returns a function that is defined as an anonymous method:

```
function MakeAdder(y: Integer): TFuncOfInt;
begin
  Result := { start anonymous method } function(x: Integer) : Integer
  begin
    Result := x + y;
  end; { end anonymous method }
end;
```

The function MakeAdder returns a function that it declares with no name: an anonymous method.

Note that MakeAdder returns a value of type TFuncOfInt. An anonymous method type is declared as a reference to a method:

```
type
  TFuncOfInt = reference to function(x: Integer): Integer;
```

This declaration indicates that the anonymous method:

- is a function
- takes one integer parameter
- returns an integer value.

In general, an anonymous function type is declared for either a procedure or function:

```
type
  TType1 = reference to procedure (parameterlist);
  TType2 = reference to function (parameterlist): returntype;
```

where *(parameterlist)* are optional.

Here are a couple of examples of types:

```
type
  TSimpleProcedure = reference to procedure;
  TSimpleFunction = reference to function(x: string): Integer;
```

An anonymous method is declared as a procedure or function without a name:

```
// Procedure
procedure (parameters)
begin
  { statement block }
end;
// Function
function (parameters): returntype
begin
  { statement block }
end;
```

where *(parameters)* are optional.

Using Anonymous Methods

Anonymous methods are typically assigned to something, as in these examples:

```
myFunc := function(x: Integer): string
begin
  Result := IntToStr(x);
end;

myProc := procedure(x: Integer)
begin
  Writeln(x);
end;
```

Anonymous methods may also be returned by functions or passed as values for parameters when calling methods. For instance, using the anonymous method variable myFunc defined just above:

```
type
  TFuncOfIntToString = reference to function(x: Integer): string;

procedure AnalyzeFunction(proc: TFuncOfIntToString);
begin
  { some code }
end;

// Call procedure with anonymous method as parameter
// Using variable:
AnalyzeFunction(myFunc);

// Use anonymous method directly:
AnalyzeFunction(function(x: Integer): string
begin
  Result := IntToStr(x);
end;)
```

Method references can also be assigned to methods as well as anonymous methods. For example:

```
type
  TMethRef = reference to procedure(x: Integer);
TMyClass = class
  procedure Method(x: Integer);
end;

var
  m: TMethRef;
  i: TMyClass;
begin
  // ...
  m := i.Method;   //assigning to method reference
end;
```

However, the converse is not true: you cannot assign an anonymous method to a regular method pointer. Method references are managed types, but method pointers are unmanaged types. Thus, for type-safety reasons, assigning method references to method pointers is not supported. For instance, events are method pointer-valued properties, so you cannot use an anonymous method for an event. See the section on variable binding for more information on this restriction.

Anonymous Methods Variable Binding

A key feature of anonymous methods is that they may reference variables that are visible to them where they were defined. Furthermore, these variables can be bound to values and wrapped up with a reference to the anonymous method. This captures state and extends the lifetime of variables.

Variable Binding Illustration

Consider again the function defined above:

```
function MakeAdder(y: Integer): TFuncOfInt;
begin
  Result := function(x: Integer): Integer
  begin
    Result := x + y;
  end;
end;
```

We can create an instance of this function that binds a variable value:

```
var
  adder: TFuncOfInt;
begin
  adder := MakeAdder(20);
  Writeln(adder(22)); // prints 42
end.
```

The variable adder contains an anonymous method that binds the value 20 to the variable y referenced in the anonymous method's code block. This binding persists even if the value goes out of scope.

Anonymous Methods as Events

A motivation for using method references is to have a type that can contain a bound variables, also known as closure values. Since closures close over their defining environment, including any local variables referenced at the point of definition, they have state that must be freed. Method references are managed types (they are reference counted), so they can keep track of this state and free it when necessary. If a method reference or closure could be freely assigned to a method pointer, such as an event, then it would be easy to create ill-typed programs with dangling pointers or memory leaks.

Delphi events are a convention for properties. There is no difference between an event and a property, except for the kind of type. If a property is of a method pointer type, then it is an event.

If a property is of a method reference type, then it should logically be considered an event too. However the IDE does not treat it as an event. This matters for classes that are installed into the IDE as components and custom controls.

Therefore, to have an event on a component or custom control that can be assigned to using a method reference or a closure value, the property must be of a method reference type. However, this is inconvenient, because the IDE does not recognize it as an event.

Here is an example of using a property with a method reference type, so it can operate as an event:

```
type
  TProc = reference to procedure;
  TMyComponent = class(TComponent)
  private
    FMyEvent: TProc;
  public
    // MyEvent property serves as an event:
    property MyEvent: TProc read FMyEvent write FMyEvent;
    // some other code invokes FMyEvent as usual pattern for events
  end;

// …

var
  c: TMyComponent;
begin
  c := TMyComponent.Create(Self);
  c.MyEvent := procedure
  begin
    ShowMessage('Hello World!'); // shown when TMyComponent invokes MyEvent
  end;
end;
```

Variable Binding Mechanism

To avoid creating memory leaks, it is useful to understand the variable binding process in greater detail.

Local variables defined at the start of a procedure, function or method (hereafter "routine") normally live only as long as that routine is active. Anonymous methods can extend these variables' lifetimes.

If an anonymous method refers to an outer local variable in its body, that variable is "captured". Capturing means extending the lifetime of the variable, so that it lives as long as the anonymous method value, rather than dying with its declaring routine. Note that variable capture captures *variables*--not *values*. If a variable's value changes after being captured by constructing an anonymous method, the value of the variable the anonymous method captured changes too, because they are the same variable with the same storage. Captured variables are stored on the heap, not the stack.

Anonymous method values are of the *method reference* type, and are reference-counted. When the last method reference to a given anonymous method value goes out of scope, or is cleared (initialized to nil) or finalized, the variables it has captured finally go out of scope.

This situation is more complicated in the case of multiple anonymous methods capturing the same local variable. To understand how this works in all situations, it is necessary to be more precise about the mechanics of the implementation.

Whenever a local variable is captured, it is added to a "frame object" associated with its declaring routine. Every anonymous method declared in a routine is converted into a method on the frame object associated with its containing routine. Finally, any frame object created because of an anonymous method value being constructed or variable being captured is chained to its parent frame by another reference--if any such frame exists and if necessary to access a captured outer variable. These links from one frame object to its parent are also reference counted. An anonymous method declared in a nested, local routine that captures variables from its parent routine keeps that parent frame object alive until it itself goes out of scope.

For example, consider this situation:

```
type
  TProc = reference to procedure;
procedure Call(proc: TProc);
// ...
procedure Use(x: Integer);
// ...

procedure L1; // frame F1
var
  v1: Integer;

  procedure L2; // frame F1_1
  begin
    Call(procedure // frame F1_1_1
    begin
      Use(v1);
    end);
  end;

begin
  Call(procedure // frame F1_2
  var
    v2: Integer;
  begin
    Use(v1);
    Call(procedure // frame F1_2_1
    begin
      Use(v2);
    end);
  end);
end;
```

Each routine and anonymous method is annotated with a frame identifier to make it easier to identify which frame object links to which:

- v1 is a variable in F1
- v2 is a variable in F1_2 (captured by F1_2_1)
- anonymous method for F1_1_1 is a method in F1_1
- F1_1 links to F1 (F1_1_1 uses v1)
- anonymous method for F1_2 is a method in F1
- anonymous method for F1_2_1 is a method in F1_2

Frames F1_2_1 and F1_1_1 do not need frame objects, since they neither declare anonymous methods nor have variables that are captured. They are not on any path of parentage between a nested anonymous method and an outer captured variable either. (They have implicit frames stored on the stack.)

Given only a reference to the anonymous method F1_2_1, variables v1 and v2 are kept alive. If instead, the only reference that outlives the invocation of F1 is F1_1_1, only variable v1 is kept alive.

It is possible to create a cycle in the method reference/frame link chains that causes a memory leak. For example, storing an anonymous method directly or indirectly in a variable that the anonymous method itself captures creates a cycle, causing a memory leak.

Utility of Anonymous Methods

Anonymous methods offer more than just a simple pointer to something that is callable. They provide several advantages:

- binding variable values
- easy way to define and use methods
- easy to parameterize using code

Variable Binding

Anonymous methods provide a block of code along with variable bindings to the environment in which they are defined, even if that environment is not in scope. A pointer to a function or procedure cannot do that.

For instance, the statement adder := MakeAdder(20); from the code sample above produces a variable adder that encapsulates the binding of a variable to the value 20.

Some other languages that implement such a construct refer to them as *closures*. Historically, the idea was that evaluating an expression like adder := MakeAdder(20); produced a closure. It represents an object that contains references to the bindings of all variables referenced in the function and defined outside it, thus closing it by capturing the values of the variables.

Ease of Use

The following sample shows a typical class definition to define some simple methods and then invoke them:

```
type
  TMethodPointer = procedure of object; // delegate void TMethodPointer();
  TStringToInt = function(x: string): Integer of object;

TObj = class
  procedure HelloWorld;
  function GetLength(x: string): Integer;
end;

procedure TObj.HelloWorld;
begin
  Writeln('Hello World');
end;

function TObj.GetLength(x: string): Integer;
begin
  Result := Length(x);
end;

var
  x: TMethodPointer;
  y: TStringToInt;
  obj: TObj;

begin
  obj := TObj.Create;

  x := obj.HelloWorld;
  x;
  y := obj.GetLength;
  Writeln(y('foo'));
end.
```

Contrast this to the same methods defined and invoked using anonymous methods:

```
type
  TSimpleProcedure = reference to procedure;
  TSimpleFunction = reference to function(x: string): Integer;

var
  x1: TSimpleProcedure;
  y1: TSimpleFunction;

begin
  x1 := procedure
    begin
      Writeln('Hello World');
    end;
  x1;   //invoke anonymous method just defined

  y1 := function(x: string): Integer
    begin
      Result := Length(x);
    end;
  Writeln(y1('bar'));
end.
```

Notice how much simpler and shorter the code is that uses anonymous methods. This is ideal if you want to explicitly and simply define these methods and use them immediately without the overhead and effort of creating a class that may never be used anywhere else. The resulting code is easier to understand.

Using Code for a Parameter

Anonymous methods make it easier to write functions and structures parameterized by code, not just values.

Multithreading is a good application for anonymous methods. if you want to execute some code in parallel, you might have a parallel-for function that looks like this:

```
type
  TProcOfInteger = reference to procedure(x: Integer);

procedure ParallelFor(start, finish: Integer; proc: TProcOfInteger);
```

The ParallelFor procedure iterates a procedure over different threads. Assuming this procedure is implemented correctly and efficiently using threads or a thread pool, it could then be easily used to take advantage of multiple processors:

```
procedure CalculateExpensiveThings;
var
  results: array of Integer;
begin
  SetLength(results, 100);
  ParallelFor(Low(results), High(results),
    procedure(i: Integer)                          // \
    begin                                          //  \ code block
      results[i] := ExpensiveCalculation(i);       //  /  used as parameter
    end                                            // /
    );
  // use results
  end;
```

Contrast this to how it would need to be done without anonymous methods: probably a "task" class with a virtual abstract method, with a concrete descendant for ExpensiveCalculation, and then adding all the tasks to a queue--not nearly as natural or integrated.

Here, the "parallel-for" algorithm is the abstraction that is being parameterized by code. In the past, a common way to implement this pattern is with a virtual base class with one or more abstract methods; consider the TThread class and its abstract Execute method. However, anonymous methods make this pattern--parameterizing of algorithms and data structures using code--far easier.

Classes and Objects Index

This section describes the object-oriented features of the Delphi language, such as the declaration and usage of class types.

Topics

Classes and Objects (Delphi)

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, fields of classes 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 objects, that 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 program outside of the object itself, a 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 do not 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 following form:

```
type
   className = class [abstract | sealed] (ancestorClass)
       type
         nestedTypeDeclaration
       const
         nestedConstDeclaration
       memberList
   end;
```

Required elements of the class type declaration

- `className` is any valid identifier.
- `memberList` declares members of the class: fields, methods, and properties.

Optional elements of the class type declaration

- `abstract`. An entire class can be declared abstract even if it does not contain any abstract virtual methods.
- `sealed`. A sealed class cannot be extended through inheritance.
- `ancestorClass`. The new class inherits directly from the predefined System.TObject class, in case you omit (ancestorClass). If you include (ancestorClass), and memberList is empty, you can omit end.
- `nestedTypeDeclaration`. Nested types present a way to keep conceptually related types together and to avoid name collisions.
- `nestedConstDeclaration`. Nested constants present a way to keep conceptually related types together and to avoid name collisions.

A class cannot be both `abstract` and `sealed`. The `[abstract | sealed]` syntax (the `[ ]` brackets and the `|` pipe between them) is used to specify that only one of the optional `sealed` or `abstract` keywords can be used. Only the `sealed` or `abstract` keywords are meaningful. The brackets and pipe symbols should be deleted.

> **Note:** Delphi allows instantiating a class declared **abstract**, for backward compatibility, but this feature should not be used anymore.

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:

```
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(const NewSize: Int64); override;
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint; override;
    function Write(const Buffer: TBytes; Offset, Count: Longint): Longint;
override;
  end; // deprecated 'Use TBytesStream';
```

`Classes.TMemoryStream` descends from `Classes.TCustomMemoryStream`, 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 `System.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;

begin
  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 Vcl.Controls.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 Vcl.Controls.TControl and in each of the Vcl.Controls.TControl 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 System.TObject class, declared in the System unit, is the ultimate ancestor of all other classes. System.TObject defines only a handful of methods, including a basic constructor and destructor. In addition to System.TObject, the System unit declares the class reference type System.TClass:

```
  TClass = class of TObject;
```

If the declaration of a class type does not specify an ancestor, the class inherits directly from System.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 Delphi compiler allows an alternative syntax to class types. 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 System.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.

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 do not 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 be 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 property in a descendent 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** properties cannot become **public** in a descendent 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 each class access to the **private** members of another class without making those members more widely accessible. For a member to be visible only inside its class, it needs to be declared **strict private**.

A **protected** member is visible anywhere in the module where its class is declared and from any descendent class, regardless of the module where the descendent 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.

Strict Visibility Specifiers

In addition to **private** and **protected** visibility specifiers, the Delphi compiler supports additional visibility settings with greater access constraints. These settings 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 descendent class, regardless of where it is declared. Furthermore, when instance members

(those declared without the **class** or **class var** keywords) are declared **strict private** or **strict protected**, they are inaccessible outside of the instance of a class in which they appear. An instance of a class cannot access **strict private** or **strict protected** instance members in other instances of the same class.

> **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 run-time 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 from 0 through 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** will not 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 Classes.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 . 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 System.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 (Delphi)

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. For example, the following declaration creates a class called TNumber whose only member, other than the methods inherited from System.TObject, is an integer field called Int:

```
type
  TNumber = class
    var
      Int: Integer;
  end;
```

The **var** keyword is optional. However, if it is not used, then all field declarations must occur before any property or method declarations. After any property or method declarations, the **var** may be used to introduce any additional field declarations.

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.

Declarations of constants and typed constants 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

Class fields are data fields in a class that can be accessed without an object reference (unlike the normal "instance fields" which are discussed above). The data stored in a class field are shared by all instances of the class and may be accessed by referring to the class or to a variable that represents an instance of the class.

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** or **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;
        var              // Ends the class var block.
          InstanceField: Integer;
    end;
```

The class fields Red, Green, and Blue can be accessed with the code:

```
TMyClass.Red := 1;
TMyClass.Green := 2;
TMyClass.Blue := 3;
```

Class fields may also be accessed through an instance of the class. With the following declaration:

```
var
       myObject: TMyClass;
```

This code has the same effect as the assignments to Red, Green, and Blue above:

```
myObject.Red := 1;
myObject.Green := 2;
myObject.Blue := 3;
```

Methods (Delphi)

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:

- Method 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**;
- warning is **platform**, **deprecated**, or **library**. For more information about these warning (hinting) directives, see Hinting Directives.

All Delphi directives are listed in Directives.

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 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 Operators" in Class References.

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 descendent 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 descendent classes. When an overridden method is called, the actual (run-time) type of the class or object used in the method call--not the declared type of the variable--determines 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 descendent 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 run time:

```
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.

Final Methods

The Delphi compiler also supports the concept of **final** virtual and dynamic methods. Declarations of final methods have the form:

```
function|procedure FunctionName; virtual|dynamic; final;
```

Here the `virtual|dynamic` syntax (two keywords and the `|` pipe between them) is used to specify that one and only one of the **virtual** or **dynamic** keywords should be used. Meaningful is only the **virtual** or **dynamic** keyword; the pipe symbol itself should be deleted.

When the keyword **final** is applied to a virtual or dynamic method, no descendent 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 compiler hints that allow it to optimize the code it produces.

> **Note:** The **virtual** or **dynamic** keywords must be written before the **final** keyword.

Example

```
type
  Base = class
    procedure TestProcedure; virtual;
    procedure TestFinalProcedure; virtual; final;
  end;

  Derived = class(Base)
    procedure TestProcedure; override;
       //Ill-formed: E2352 Cannot override a final method
    procedure TestFinalProcedure; override;
  end;
```

Virtual versus Dynamic

In Delphi for Win32, virtual and dynamic methods are semantically equivalent. However, they differ in the implementation of method-call dispatching at run time: 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 that are inherited by many descendent 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 does not include **override**, the new declaration merely hides the inherited one without overriding it. Both methods exist in the descendent 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 descendent 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.

Class Methods

Most methods are called instance methods, because they operate on an individual instance of an object. A class method is a method (other than a constructor) that operates on classes instead of objects. There are two types of class methods: ordinary class methods and class static methods.

Ordinary Class Methods

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 can 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 instance fields, instance properties, and normal (object) methods. You can use **Self** to call constructors and other class methods, or to access class properties and class fields.

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**.

Class Static Methods

Like class methods, class static methods can be accessed without an object reference. Unlike ordinary class methods, class static methods have no **Self** parameter at all. They also cannot access any instance members. (They still have access to class fields, class properties, and class methods.) Also unlike class methods, 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;
```

Like a class method, you can call a class static method through the class type (for example, without having an object reference), such as:

```
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 descendent 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 descendent 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 run time 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 in Procedures and Functions (Delphi).

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 the 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 descendent 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 **virtual** is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objects--that is, construction of objects whose types are not known at compile time. (See Class References.)

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 implementation is typically to call the inherited destructor to destroy the inherited fields of the object.

When an exception is raised during the 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 to check for **nil** values before destroying an object.

Class Constructors

A class constructor is a special class method that is not accessible to developers. Calls to class constructors are inserted automatically by the compiler into the initialization section of the unit where the class is defined. Normally, class constructors are used to initialize the static fields of the class or to perform a type of initialization, which is required before the class or any class instance can function properly. Even though the same result can be obtained by placing class initialization code into the *initialization* section, class constructors have the benefit of helping the compiler decide which classes should be included into the final binary file and which should be removed from it.

The next example shows the usual way of initializing class fields:

```
type
  TBox = class
  private
    class var FList: TList<Integer>;
  end;

implementation

initialization
  { Initialize the static FList member }
  TBox.FList := TList<Integer>.Create();

end.
```

This method has a big disadvantage: even though an application can include the unit in which **TBox** is declared, it may never actually use the **TBox** class. In the current example, the TBox class is included into the resulting binary, because it is referenced in the initialization section. To alleviate this problem, consider using class constructors:

```
type
  TBox = class
  private
    class var FList: TList<Integer>;
    class constructor Create;
  end;

implementation

class constructor TBox.Create;
begin
  { Initialize the static FList member }
  FList := TList<Integer>.Create();
end;

end.
```

In this case, the compiler checks whether TBox is actually used anywhere in the application, and if it is used, a call to the class constructor is added automatically to the initialization section of the unit.

Note: Even though the compiler takes care of ordering the initialization of classes, in some complex scenarios, ordering may become random. This happens when the class constructor of a class depends on the state of another class that, in turn, depends on the first class.

Note: The class constructor for a generic class or record may execute multiple times. The exact number of times the class constructor is executed in this case depends on the number of specialized versions of the generic type. For example, the class constructor for a specialized *TList<String>* class may execute multiple times in the same application.

Class Destructors

Class destructors are the opposite of class constructors in that they perform the finalization of the class. Class destructors come with the same advantages as class constructors, except for *finalization* purposes.

The following example builds on the example shown in class constructors and introduces the finalization routine:

```
type
  TBox = class
  private
    class var FList: TList<Integer>;
    class constructor Create;
    class destructor Destroy;
  end;

implementation

class constructor TBox.Create;
begin
  { Initialize the static FList member }
  FList := TList<Integer>.Create();
end;

class destructor TBox.Destroy;
begin
  { Finalize the static FList member }
  FList.Free;
end;

end.
```

Note: The class destructor for a generic class or record may execute multiple times. The exact number of times the class destructor is executed in this case depends on the number of specialized versions of the generic type. For example, the class destructor for a specialized *TList<String>* class may execute multiple times in the same application.

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 from 1 through 49151 that 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 does not have to specify the same method name or parameter type as the method it overrides. The message ID alone determines to which message the method responds and whether it is an override.

Implementing Message Methods

The implementation of a message method can call the inherited message method, as in the following 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 (Delphi)

This topic describes the following material:

- Property access
- Array properties
- Index specifiers
- Storage specifiers
- Property overrides and redeclarations
- Class properties

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 with a given signature (array parameter list), but it is possible to overload the default property. Changing or hiding the default property in descendent 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 does not 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 is not `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 descendent 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

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 is declared with the **class property** keywords. Class properties cannot be **published**, and cannot have **stored** or **default** value definitions.

You can introduce a block of class static 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
     strict private
       class var         // Note fields must be declared as class fields
          FRed: Integer;
          FGreen: Integer;
          FBlue: Integer;
       public             // ends the class var block
          class property Red: Integer read FRed write FRed;
          class property Green: Integer read FGreen write FGreen;
          class property Blue: Integer read FBlue write FBlue;
   end;
```

You can access the above class properties with the code:

```
TMyClass.Red := 0;
TMyClass.Blue := 0;
TMyClass.Green := 0;
```

Events (Delphi)

This topic describes the following material:

- Event properties and event handlers
- Triggering multiple event handlers

About Events

An event links an occurrence in the system with the code that responds to that occurrence. The occurrence triggers the execution of a procedure called an event handler. The event handler performs the tasks that are required in response to the occurrence. Events allow the behavior of a component to be customized at design-time or at run time. To change the behavior of the component, replace the event handler with a custom event handler that will have the desired behavior.

Event Properties and Event Handlers

Components that are written in Delphi use properties to indicate the event handler that will be executed when the event occurs. By convention, the name of an event property begins with "On", and the property is implemented with a field rather than read/write methods. The value stored by the property is a method pointer, pointing to the event handler procedure.

In the following example, the TObservedObject class includes an OnPing event, of type TPingEvent. The FOnPing field is used to store the event handler. The event handler in this example, TListener.Ping, prints 'TListener has been pinged!'.

```
program EventDemo;

{$APPTYPE CONSOLE}
type
  { Define a procedural type }
  TPingEvent = procedure of object;

  { The observed object }
  TObservedObject = class
  private
    FPing: TPingEvent;

  public
    property OnPing: TPingEvent read FPing write FPing;

    { Triggers the event if anything is registered }
    procedure TriggerEvent();
  end;

  { The listener }
  TListener = class
    procedure Ping;
  end;


procedure TObservedObject.TriggerEvent;
begin
  { Call the registerd event only if there is a listener }
  if Assigned(FPing) then
    FPing();
end;

procedure TListener.Ping;
begin
  Writeln('TListener has been pinged.');
end;

var
  ObservedObject: TObservedObject;
  Listener: TListener;

begin
  { Create object instances }
  ObservedObject := TObservedObject.Create();
  Listener := TListener.Create();

  { Register the event handler }
  ObservedObject.OnPing := Listener.Ping;

  { Trigger the event }
  ObservedObject.TriggerEvent();//Should output 'TListener has been pinged'
  Readln;                       // Pause console before closing
end.
```

Triggering Multiple Event Handlers

In Delphi, events can be assigned only a single event handler. If multiple event handlers must be executed in response to an event, the event handler assigned to the event must call any other event handlers. In the following code, a subclass of TListener called TListenerSubclass has its own event handler called Ping2. In this example, the Ping2 event handler must explicitly call the TListener.Ping event handler in order to trigger it in response to the OnPing event:

```
program EventDemo2;

{$APPTYPE CONSOLE}

type
  { Define a procedural type }
  TPingEvent = procedure of object;

  { The observed object }
  TObservedObject = class
  private
    FPing: TPingEvent;

  public
    property OnPing: TPingEvent read FPing write FPing;

    { Triggers the event if anything is registered }
    procedure TriggerEvent();
  end;

  { The listener }
  TListener = class
    procedure Ping;
  end;

  { The listener sub-class }
  TListenerSubclass = class(TListener)
    procedure Ping2;
  end;

procedure TObservedObject.TriggerEvent;
begin
  { Call the registerd event only if there is a listener }
  if Assigned(FPing) then
    FPing();
end;

procedure TListener.Ping;
begin
  Writeln('TListener has been pinged.');
end;

procedure TListenerSubclass.Ping2;
begin
  { Call the base class ping }
  Self.Ping();
  Writeln('TListenerSubclass has been pinged.');
end;

var
  ObservedObject: TObservedObject;
```

```
  Listener: TListenerSubclass;

begin
  { Create object instances }
  ObservedObject := TObservedObject.Create();
  Listener := TListenerSubclass.Create();

  { Register the event handler }
  ObservedObject.OnPing := Listener.Ping2;

  { Trigger the event }
  ObservedObject.TriggerEvent();//Should output 'TListener has been pinged'
                               //and then 'TListenerSubclass has been pinged'
  Readln;                      // Pause console before closing
end.
```

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 sometimes 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-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 **System.Classes.TCollection** (in the **System.Classes** unit):

```
type TCollectionItemClass = class of TCollectionItem;
     ...
TCollection = class(TPersistent)
     ...
  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 constructor of the class. 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

Class methods operate on class references. Every class inherits two class methods from TObject, called ClassType and ClassParent. These methods return, respectively, a reference to the class of an object and to the immediate ancestor class of an object. 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 other -- a compilation error results. For example:

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

This statement casts the `ActiveControl` variable to the TEdit type. First it verifies that the object referenced by `ActiveControl` 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 run time, *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';
```

Code Examples

- ClassParent (Delphi)

Exceptions (Delphi)

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 SysUtils.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 SysUtils.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 SysUtils.EMathError exception handler that also handles SysUtils.EInvalidOp, SysUtils.EZeroDivide, SysUtils.Overflow, and SysUtils.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 SysUtils.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. SysUtils.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 SysUtils.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. SysUtils.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 SysUtils.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 SysUtils.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

SysUtils.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 SysUtils.Exception.

The SysUtils.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.

Class and Record Helpers (Delphi)

This topic describes the syntax of class helper declarations.

About Class and Record Helpers

A class or a record helper is a type that - when associated with another class or a record - introduces additional method names and properties that may be used in the context of the associated type (or its descendants). Helpers are a way to extend a class without using inheritance, which is also useful for records that do not allow inheritance at all. A helper simply introduces a wider scope for the compiler to use when resolving identifiers. When you declare a class or a record helper, you state the helper name, and the name of the type you are going to extend with the helper. You can use the helper any place where you can legally use the extended class or record. The compiler's resolution scope then becomes the original type, plus the helper.

Class and record helpers provide a way to extend a type, but they should not be viewed as a design tool to be used when developing new code. For new code you should always rely on normal class inheritance and interface implementations.

Helper Syntax

The syntax for declaring a class helper is:

```
type
   identifierName = class|record helper [(ancestor list)] for
TypeIdentifierName
     memberList
   end;
```

The *ancestor list* is optional. It can be specified only for class helper.

A helper type may not declare instance data, but class fields are allowed.

The visibility scope rules and *memberList* syntax are identical to that of ordinary class and record types.

Note: Class and record helpers do not support operator overloading.

You can define and associate multiple helpers with a single type. However, only zero or one helper applies in any specific location in source code. The helper defined in the nearest scope will apply. Class or record helper scope is determined in the normal Delphi fashion (for example, right to left in the unit's **uses** clause).

Using Helpers

The following code demonstrates the declaration of a class helper (record helpers behave in the same manner):

```
type
   TMyClass = class
      procedure MyProc;
      function  MyFunc: Integer;
   end;

   ...

   procedure TMyClass.MyProc;
   var X: Integer;
   begin
      X := MyFunc;
   end;

   function TMyClass.MyFunc: Integer;
   begin
       ...
   end;

...

type
   TMyClassHelper = class helper for TMyClass
     procedure HelloWorld;
     function MyFunc: Integer;
   end;

   ...

   procedure TMyClassHelper.HelloWorld;
   begin
      Writeln(Self.ClassName); // Self refers to TMyClass type, not
TMyClassHelper
   end;

   function TMyClassHelper.MyFunc: Integer;
   begin
     ...
   end;

...

var
  X: TMyClass;
begin
  X := TMyClass.Create;
  X.MyProc;    // Calls TMyClass.MyProc
  X.HelloWorld; // Calls TMyClassHelper.HelloWorld
  X.MyFunc;    // Calls TMyClassHelper.MyFunc
```

Note that the class helper function MyFunc is called, because the class helper takes precedence over the actual class type.

Nested Type Declarations

Type declarations can be nested within class declarations. Nested types are used 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

The *nestedTypeDeclaration* follows the type declaration syntax defined in Data Types, Variables, and Constants Index (Delphi).

```
type
  className = class [abstract | sealed] (ancestorType)
      memberList

      type
         nestedTypeDeclaration

      memberList
  end;
```

Nested type declarations are terminated by the first occurrence 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. 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 example demonstrates 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, specifically, reserved words or visibility specifiers. Typed constants are not supported, so you cannot declare nested constants of value types, such as System.Currency, or System.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.
```

Operator Overloading (Delphi)

This topic describes Delphi's operator methods and how to overload them.

About Operator Overloading

Delphi allows certain functions, or "operators", to be overloaded within record declarations. The name of the operator function maps to a symbolic representation in source code. For example, the Add operator maps to the **+** symbol.

The compiler generates a call to the appropriate overload, matching the context (that is, the return type, and type of parameters used in the call), to the signature of the operator function.

The following table shows the Delphi operators that can be overloaded:

Operator	Category	Declaration Signature	Symbol Mapping
Implicit	Conversion	Implicit(a : type) : resultType;	implicit typecast
Explicit	Conversion	Explicit(a: type) : resultType;	explicit typecast
Negative	Unary	Negative(a: type) : resultType;	**-**
Positive	Unary	Positive(a: type): resultType;	**+**
Inc	Unary	Inc(a: type) : resultType;	Inc
Dec	Unary	Dec(a: type): resultType	Dec
LogicalNot	Unary	LogicalNot(a: type): resultType;	**not**
Trunc	Unary	Trunc(a: type): resultType;	Trunc
Round	Unary	Round(a: type): resultType;	Round
In	Set	In(a: type; b: type) : Boolean;	**in**
Equal	Comparison	Equal(a: type; b: type) : Boolean;	**=**
NotEqual	Comparison	NotEqual(a: type; b: type): Boolean;	**<>**
GreaterThan	Comparison	GreaterThan(a: type; b: type) Boolean;	**>**
GreaterThanOrEqual	Comparison	GreaterThanOrEqual(a: type; b: type): Boolean;	**>=**
LessThan	Comparison	LessThan(a: type; b: type): Boolean;	**<**
LessThanOrEqual	Comparison	LessThanOrEqual(a: type; b: type): Boolean;	**<=**
Add	Binary	Add(a: type; b: type): resultType;	**+**
Subtract	Binary	Subtract(a: type; b: type) : resultType;	**-**
Multiply	Binary	Multiply(a: type; b: type) : resultType;	*****
Divide	Binary	Divide(a: type; b: type) : resultType;	**/**
IntDivide	Binary	IntDivide(a: type; b: type): resultType;	**div**

Modulus	Binary	Modulus(a: type; b: type): resultType;	**mod**
LeftShift	Binary	LeftShift(a: type; b: type): resultType;	**shl**
RightShift	Binary	RightShift(a: type; b: type): resultType;	**shr**
LogicalAnd	Binary	LogicalAnd(a: type; b: type): resultType;	**and**
LogicalOr	Binary	LogicalOr(a: type; b: type): resultType;	**or**
LogicalXor	Binary	LogicalXor(a: type; b: type): resultType;	**xor**
BitwiseAnd	Binary	BitwiseAnd(a: type; b: type): resultType;	**and**
BitwiseOr	Binary	BitwiseOr(a: type; b: type): resultType;	**or**
BitwiseXor	Binary	BitwiseXor(a: type; b: type): resultType;	**xor**

No operators other than those listed in the table may be defined on a class or record.

Overloaded operator methods cannot be referred to by name in source code. To access a specific operator method of a specific class or record, refer to: Code Example:OpOverloads_(Delphi). Operator identifiers are included for classes and records in the language in the class or record's list of methods starting with the word "operator" (example: System.AnsiStringBase Methods). You can implement any of the above operators in your own classes and records.

The compiler will use an operator for a class or record provided that:

- For binary operators, one of the input parameters must be the class type.
- For unary operators, either the input parameter or the return value must be the class type.
- For a logical operator and a bitwise operator using the same symbol, the logical operator is used only when the operands are booleans. Since the type of the class of this class operator is not a boolean, a logical operator will only be used when the other operand is a boolean.

No assumptions are made regarding the distributive or commutative properties of the operation. For binary operators, the first parameter is always the left operand, and the second parameter is always the right operand. Associativity is assumed to be left-to-right in the absence of explicit parentheses.

Resolution of operator methods is done over the union of accessible operators of the types used in the operation (note this includes inherited operators). For an operation involving two different types A and B, if type A has an implicit conversion to B, and B has an implicit conversion to A, an ambiguity will occur. Implicit conversions should be provided only where absolutely necessary, and reflexivity should be avoided. It is best to let type B implicitly convert itself to type A, and let type A have no knowledge of type B (or vice versa).

As a general rule, operators should not modify their operands. Instead, return a new value, constructed by performing the operation on the parameters.

Overloaded operators are used most often in records (that is, value types).

Note: Class and record helpers do not support operator overloading.

Declaring Operator Overloads

Operator overloads are declared within classes or records, with the following syntax:

```
type
   typeName = record
       class operator conversionOp(a: type): resultType;
       class operator unaryOp(a: type): resultType;
       class operator comparisonOp(a: type; b: type): Boolean;
       class operator binaryOp(a: type; b: type): resultType;
   end;
```

Implementation of overloaded operators must also include the **class operator** syntax:

```
class operator typeName.conversionOp(a: type): resultType;
class operator typeName.unaryOp(a: type): resultType;
class operator typeName.comparisonOp(a: type; b: type): Boolean;
class operator typeName.binaryOp(a: type; b: type): resultType;
```

The following are some examples of overloaded operators:

```
type
   TMyRecord = record
     class operator Add(a, b: TMyRecord): TMyRecord;      // Addition of two
operands of type TMyRecord
     class operator Subtract(a, b: TMyRecord): TMyRecord; // Subtraction of
type TMyRecord
     class operator Implicit(a: Integer): TMyRecord;      // Implicit
conversion of an Integer to type TMyRecord
     class operator Implicit(a: TMyRecord): Integer;      // Implicit
conversion of TMyRecordto Integer
     class operator Explicit(a: Double): TMyRecord;       // Explicit
conversion of a Double to TMyRecord
   end;

// Example implementation of Add
class operator TMyRecord.Add(a, b: TMyRecord): TMyRecord;
begin
   // ...
end;

var
x, y: TMyRecord;
begin
   x := 12;      // Implicit conversion from an Integer
   y := x + x;   // Calls TMyRecord.Add(a, b: TMyRecord): TMyRecord
   b := b + 100; // Calls TMyRecord.Add(b, TMyRecord.Implicit(100))
end;
```

Code Samples

- RTL.ComplexNumbers Sample

Standard Routines and Input-Output

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.

> **Note:** For new programs, you might want to use the File Management classes and functions in the System.Classes and System.SysUtils units. System.Classes.TStream and its descendent classes are currently recommended for general file handling in Delphi (for related routines, see Streams, Reader and Writers). For text-file handling, TStreamReader and TStreamWriter are recommended over calling Write and Writeln. API Categories Index contains lists of related routines and classes.
>
> **Note:** BlockRead and BlockWrite have untyped parameters, which can be the source of memory corruption. Both methods depend on the setting of record size, implicitly made by a previous call of Reset or Rewrite. Using Streams gives a greater level of flexibility and functionality to the programmer.

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. Note that file types are only available on the Win32 platform.

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 does not 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 do not 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, System.Input and System.Output The standard file variable System.Input is a read-only file associated with the operating system's standard input (typically, the keyboard). The standard file variable System.Output is a write-only file associated with the operating system's standard output (typically, the display). Before an application begins executing, System.Input and System.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 Linking 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 System.Input or System.Output will produce an I/O error.

Some of the standard I/O routines that work on text files do not need to have a file variable explicitly given as a parameter. If the file parameter is omitted, System.Input or System.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;
    CodePage := DefaultSystemCodePage;
    Name[0] := #0;
   end;
  end;
```

The CodePage field must be set to DefaultSystemCodePage for the device-interface functions to be used by the RTL. These device-interface functions must perform any special character handling that is needed.

The device-interface functions can use the UserData field in the file record to store private information. This field is not 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 does not 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.
StrFmt	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 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	Initializes 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.
sin	Returns the sine of an angle in radians.
SizeOf	Returns the number of bytes occupied by a variable or type.
Slice	Returns a sub-section of an array.
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 Index

This section describes how to create static and dynamically loadable libraries in Delphi.

> **Note:** Libraries are significantly more limited than packages in what they can export. Libraries cannot export constants, types, and normal variables. That is, class types defined in a library will not be seen in a program using that library.
>
> To export items other than simple procedures and functions, **packages** are the recommended alternative. Libraries should only be considered when interoperability with other programming is a requirement.

Topics

- Libraries and Packages (Delphi)
- Writing Dynamically Loaded Libraries
- Packages (Delphi)

Libraries and Packages (Delphi)

A dynamically loadable library is a dynamic-link library (**DLL**) on Windows, a **DYLIB** on Mac, or a shared object (**SO**) on Linux. 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 run time, to the programs that use it.

Delphi programs can call DLLs and shared objects written in other languages, and applications written in other languages can call DLLs or shared objects written in Delphi.

Calling Dynamically Loadable Libraries

You can call operating system routines that are not linked to your application. These routines are usually in a DLL or in a shared object. On Windows and OS X, there is no compile-time validation of attempts to import a routine. It means that the library does not need to be present when you compile your program.

On other platforms, such as Linux, to resolve an external reference, you have to link to a shared object. If you want to avoid validation, use `LoadLibrary` and `GetProcAddress` as described in the Dynamic Loading section.

Before you can call routines defined in DLL or shared object, 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 run time.

Delphi does not support importing variables from DLLs or shared objects.

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 the 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.

Delayed Loading (Windows-only)

The **delayed** directive can be used to decorate an *external* routine to delay the loading of the library containing the routine. The actual loading happens when the routine is called for the first time. The following example demonstrates the use of the *delayed* directive:

```
function GetSomething: Integer; external 'somelibrary.dll' delayed;
```

In the example above, the *GetSomething* routine is imported from the *somelibrary.dll* library. The delayed directive ensures that *somelibrary.dll* is not statically linked to the application, but rather dynamically.

The **delayed** directive is useful in the case where the imported routines do not exist on the target operating system on which the application is run. Statically imported routines require that the operating system find and load the library when the application is started. If the routine is not found in the loaded library, or the library does not exist, the Operating System halts the execution of the application. Using the **delayed** directive enables you to check, at run time, whether the Operating System supports the required APIs; only then you can call the imported routines.

Another potential use for the **delayed** directive is related to the memory footprint of the application: decorating the less probably to be used routines, as **delayed** may decrease the memory footprint of the application, because the libraries are loaded only when required. The abusive use of **delayed** can damage the speed performance of the program (as perceived by the end user).

Note: Trying to call a delayed routine that cannot be resolved results in a run-time error (or an exception, if the SysUtils unit is loaded).

In order to fine-tune the delay-loading process used by the Delphi Run-time Library, you can register hook procedures to oversee and change its behavior. To accomplish this, use **SetDliNotifyHook2** and **SetDliFailureHook2**, declared in the SysInit unit. Also see the code example at DelayedLoading (Delphi).

Dynamic Loading

You can access routines in a library through direct calls to Windows APIs, including LoadLibrary, FreeLibrary, and GetProcAddress. They are also available on OS X, Linux, and Android. These functions are declared in `Winapi.Windows.pas` unit for Windows and in `System.SysUtils.pas` for other platforms. In this case, use procedural-type variables to reference the imported routines.

For example:

```
uses System.SysUtils {$IFDEF MSWINDOWS},Winapi.Windows{$ENDIF};

type
   TTimeRec = record
     Second: Integer;
     Minute: Integer;
     Hour: Integer;
   end;

   TGetTime = procedure(var Time: TTimeRec);

var
  Time: TTimeRec;
  Handle: HMODULE;
  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

***Note:** Libraries are significantly more limited than packages in what they can export. Libraries cannot export constants, types, and normal variables. That is, class types defined in a library will not be seen in a program using that library. To export items other than simple procedures and functions, **packages** are the recommended alternative. Libraries should only be considered when interoperability with other programming is a requirement.*

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.

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 used in multithreading applications. 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 parameter, for 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.

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

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 (Delphi)

Packages are typically the preferred way to export items other than simple procedures and functions. Libraries should only be considered when interopability with other programming is a requirement.

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.

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` and `DATAX.DCP`

`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` 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	A binary image containing a package header and the concatenation of all .dcu (Win32) files in the package. A single .dcp or .dcp file is created for each package. The base name for the file is the base name of the .dpk source file.
BPL	The runtime package. This file is a DLL on Win32 with special RAD Studio-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 `dcp` 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.

Object Interfaces Index

This section describes the use of interfaces in Delphi.

Topics

- Object Interfaces (Delphi)
- Implementing Interfaces
- Interface References (Delphi)
- Automation Objects (Windows only)

Object Interfaces (Delphi)

An object interface, or simply interface, defines methods that can be implemented by a class. Interfaces are declared as 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 method of the interface. 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 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;
```

Warning: The `ancestorInterface` and GUID specification are required to support Win32 COM interoperability. If your interface is to be accessed through COM, be sure to specify the `ancestorInterface` and GUID.

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**.

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.

> **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 `TInterfacedObject` of the `System` unit. 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`.

Warning:

`QueryInterface`, `_AddRef`, and `_Release` are required to support Win32 COM interoperability. If your interface is to be accessed through COM, be sure to implement these methods.

Interface Identification and GUIDs

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:

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

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.

Note: GUIDs are only used for COM interoperability.

The TGUID and PGUID types, declared in the System unit, are used to manipulate GUIDs.

```
type
    PGUID = ^TGUID;
    TGUID = packed record
      D1: Cardinal;
      D2: Word;
      D3: Word;
      D4: array[0..7] of Byte;
  end;
```

Supports can be called in either of two ways:

```
if Supports(Allocator, IMalloc) then ...
```

or:

```
if Supports(Allocator, IID_IMalloc) then ...
```

> **Note:** The SysUtils unit provides an overloaded function called **Supports** that returns true or false when class types and instances support a particular interface represented by a GUID. The **Supports** function is used in the manner of the Delphi **is** and **as** operators. The significant difference is that the **Supports** function can take as the right operand either a GUID or an interface type associated with a GUID, whereas **is** and **as** take the name of a type. For more information about **is** and **as**, see Class References.

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**. 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 declaration of the class, after the name of the class 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 declaration of `System.TInterfacedObject` (on Windows. On other platforms, declaration is slightly different):

```
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`.) Every interface inherits from `IInterface`, and a class that implements interfaces must implement the `QueryInterface`, `_AddRef`, and `_Release` methods. `TInterfacedObject` in the unit `System` implements these methods and is thus a convenient base from which to derive other classes that implement interfaces.

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 the `Alloc` and `Free` methods of `IMalloc` onto the `Allocate` and `Deallocate` methods of `TMemoryManager`.

A method resolution clause cannot alter a mapping introduced by an ancestor class.

Changing Inherited Implementations

Descendent 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 descendent class' declaration. For example:

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    // ...
  end;
  TWindow = class(TInterfacedObject, IWindow)
    // TWindow implements IWindow pocedure 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 `System.TAggregatedObject`.

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 (Delphi)

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 indicate 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 descendent 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 you 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, you cannot use it to access TSomething's P2 method, since P2 is not a method of IDescendant. But if you 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 System.IInterface. 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 (run-time) type of 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 the interface to the QueryInterface method in object, raising an exception unless QueryInterface returns zero. If QueryInterface returns zero (indicating that the object's class implements the interface), the interface query returns an interface reference to object.

Casting Interface References to Objects

The **as** operator can also be used to cast an interface reference back to the object from which it was obtained. This casting only works for interfaces obtained from Delphi objects. For example:

```
var
  LIntfRef: IInterface;
  LObj: TInterfacedObject;
begin
  { Create an interfaced object and extract an interface from it. }
  LIntfRef := TInterfacedObject.Create();

  { Cast the interface back to the original object. }
  LObj := LIntfRef as TInterfacedObject;
end;
```

The above example shows how to obtain the original object from which the interface reference was obtained. This technique is useful when possessing an interface reference is simply not enough.

The **as** operator raises an exception if the interface was not extracted from the given class:

```
var
  LIntfRef: IInterface;
  LObj: TInterfacedObject;
begin
  { Create an interfaced object and extract an interface from it. }
  LIntfRef := TInterfacedObject.Create();

  try
    { Cast the interface to a TComponent. }
    LObj := LIntfRef as TComponent;
  except
    Writeln('LIntfRef was not referencing a TComponent instance');
  end;
end;
```

You can also perform normal type casting (unsafe) from an interface reference to an object. Like in the case of object unsafe casting, this method does not raise any exceptions. The difference between the unsafe object-to-object casting and unsafe interface-to-object casting is that while the first returns a valid pointer in case of incompatible types, the later returns *nil*. The example describes the use of unsafe casting:

```
var
  LIntfRef: IInterface;
  LObj: TInterfacedObject;
begin
  { Create an interfaced object and extract an interface from it. }
  LIntfRef := TInterfacedObject.Create();

  { Cast the interface to a TComponent. }
  LObj := TComponent(LIntfRef);

  if LObj = nil then
    Writeln('LIntfRef was not referencing a TComponent instance');

  { Cast the interface to a TObject. }
  LObj := TObject(LIntfRef);

  if LObj <> nil then
    Writeln('LIntfRef was referencing a TObject (or descendant).');
end;
```

To avoid potential **nil** references, use the **is** operator to verify whether the interface reference was extracted from a given class:

```
if Intf is TCustomObject then ...
```

> **Note:** Make sure you are using Delphi-only objects when using the unsafe casting or the **as** and **is** operators.

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 run time; a class cannot implement a dispatch interface.

A dispatch interface type declaration has the form:

```
type InterfaceName = dispinterface
  ['{GUID}']
  // …
end;
```

where

```
['{GUID}']
```

is optional and the interface contains 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 **readonly** or **writeonly**. 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 run time 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**, **System.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**, **System/TDateTime**, Variant, **OleVariant**, and **WordBool**.)

Memory Management Index

This section describes memory management issues related to programming in Delphi on Win32.

Topics

- Memory Management
- Internal Data Formats (Delphi)

Memory Management

This help topic describes the two memory managers that are used on the various target platforms, and briefly describes memory issues of variables.

Default memory manager

The memory manager in use is determined by the target platform/compiler of your application.

The following table lists the default memory manager for each platform.

Platform	Compiler	Memory Manager name
Win32	DCC32	FastMM (GETMEM.inc)
Win64	DCC64	FastMM (GETMEM.inc)
OSX32	DCCOSX	Posix/32
Linux	DCCLINUX64	Posix/64
iOSDevice32	DCCIOSARM	Posix/32
iOSDevice64	DCCIOSARM64	Posix/64
iOSSimulator	DCCIOS32	Posix/32
Android	DCCAARM	Posix/32

The FastMM Memory Manager (Win32 and Win64)

The Memory Manager manages all dynamic memory allocations and deallocations in an application. The New, Dispose, GetMem, ReallocMem, and

FreeMem standard System procedures use the memory manager. All objects, dynamic arrays, and long strings are allocated through the memory manager.

For **Win32** and **Win64**, the default FastMM 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. The Memory Manager is optimized for efficient operation (high speed and low memory overhead) in single and multi-threaded applications. 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 virtual memory API (the >VirtualAlloc and VirtualFree functions).

For **Win32**, the Memory Manager supports a user mode address space up to 2GB.

> **Note:** To increase the user mode address space to 3GB, see Increasing the Memory Address Space topic.

Memory Manager blocks are rounded upward to a size that is a multiple of 4 bytes, and include a 4-byte header in which the size of the block and other status bits are stored. The start address of memory blocks are aligned on at least 8-byte boundaries, or optionally on 16-byte boundaries, which improves performance when addressing them. (See System.SetMinimumBlockAlignment)

For **Win64**, the Memory Manager supports a user mode address space up to 16EiB in speculation.

> **Note:** Actual maximum allocatable size is depended on CPU implementation and operating system. For example, the current Intel/X64 implementation supports up to 256TiB (48bits), and Windows 7 Professional supports up to 192GiB.

Memory Manager blocks are rounded upward to a size that is a multiple of 16 bytes, and include a 8-byte header in which the size of the block and other status bits are stored. The start address of memory blocks are aligned on at least 16-byte boundaries.

For **Win32** and **Win64**, the Memory Manager employs an algorithm that anticipates future block reallocations, reducing the performance impact usually associated with such operations. The reallocation algorithm also helps reduce address space fragmentation. The memory manager provides a sharing mechanism that does not require the use of an external DLL.

The Memory Manager includes reporting functions to help applications monitor their own memory usage and potential memory leaks.

The Memory Manager provides two procedures, GetMemoryManagerState and GetMemoryMap, that allow applications to retrieve memory-manager status information and a detailed map of memory usage.

The Posix Memory Manager (Posix platforms)

The Posix Memory Manager is used when the target platform/compiler is OSX 32, Linux, 32-bit iOS Device, 64-bit iOS Device, iOS Simulator, or Android.

All memory management functions/methods use the Posix system library, as shown in the following table:

RTL	POSIX function
AllocMem	calloc
FreeMem	free
GetMem	malloc
ReallocMem	realloc

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 the stack of an application. 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 (Delphi)

The following topics describe the internal formats of Delphi data types.

Integer Types

Integer values have the following internal representation in Delphi.

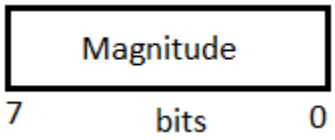
Platform-Independent Unsigned Integer Types

Values of platform-independent integer types occupy the same number of bits on any platform.

Values of unsigned integer types always are positive and do not involve a **Sign bit** as do signed integer types. All bits of unsigned integer types occupy by the magnitude of the value and have no other meaning.

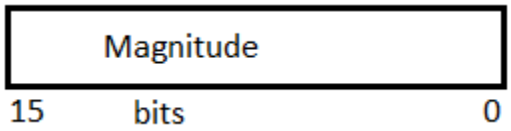
Byte, UInt8

Byte and UInt8 are 1-byte (8-bit) unsigned positive integer numbers. The **Magnitude** occupies all 8-bits.



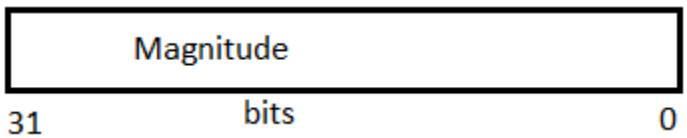
Word and UInt16

Word and UInt16 are 2-byte (16-bit) unsigned integer numbers.



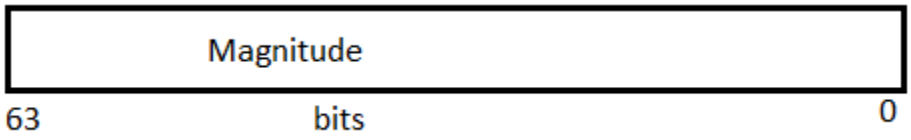
FixedUInt, Cardinal and UInt32

FixedUInt, Cardinal, and UInt32 are 4-byte (32-bit) unsigned integer numbers.



UInt64

UInt64 are 8-byte (64-bit) unsigned integer numbers.



Platform-Independent Signed Integer Types

Values of signed integer types represent a sign of a number by one leading *sign bit*, expressed by the most significant bit. The sign bit is 0 for a positive number, and 1 for a negative number. Other bits in a positive signed integer number are occupied by the magnitude. In a negative signed integer number, other bits are occupied by the *two's complement* representation of the magnitude of the value (absolute value).

To obtain the two's complement to a magnitude:

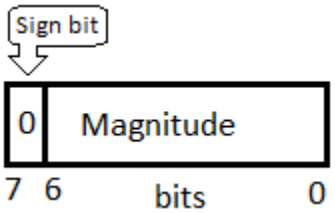
1. Starting from the right, find the first '1'.
2. Invert all of the bits to the left of that one.

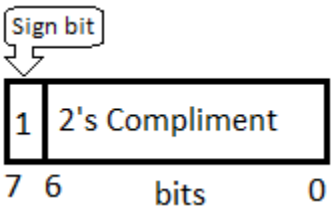
For example:

	Example 1	**Example 2**
Magnitude	0101010	1010101
2's Complement	1010110	0101011

ShortInt, Int8

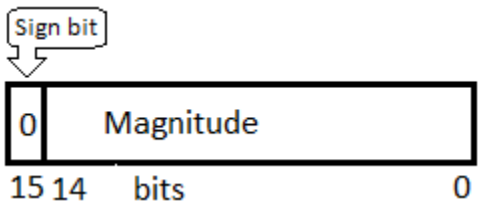
Shortint and Int8 are 1-byte (8-bit) signed integer numbers. The **sign bit'** occupies the most significant 7-th bit, the **Magnitude** or **two's complement** occupies other 7 bits.

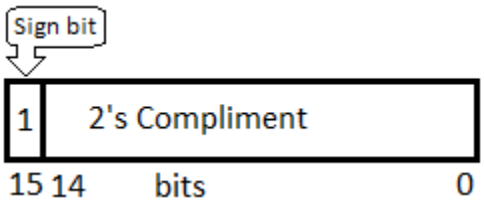




SmallInt and Int16

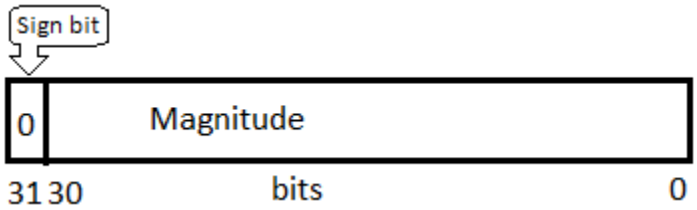
SmallInt and Int16 are 2-byte (16-bit) signed integer numbers.

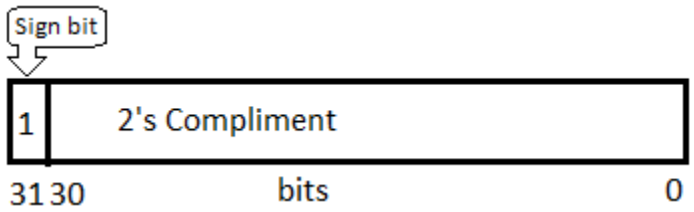




FixedInt, Integer and Int32

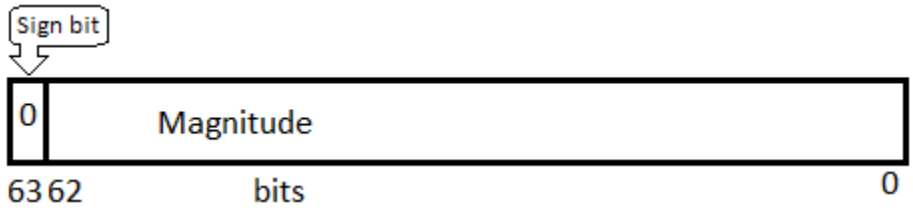
FixedInt, Integer, and Int32 are 4-byte (32-bit) signed integer numbers.

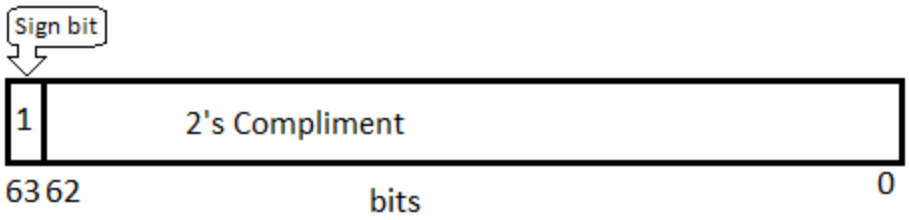




Int64

Int64 are 8-byte (64-bit) signed integer numbers.





Platform-Dependent Integer Types

The platform-dependent integer types are transformed to fit the bit size of the current target platform. On 64-bit platforms they occupy 64 bits, on 32-bit platforms they occupy 32 bits (except the LongInt and LongWord types). When the size of the target platform is the same as the CPU platform, then one platform-dependent integer number exactly matches the size of CPU registers. These types are often used when best performance is desired for a particular CPU type and operating system.

Unsigned Integer NativeUInt

NativeUInt is the platform-dependent unsigned integer type. The size and internal representation of NativeUInt depends on the current platform. On 32-bit platforms, NativeUInt is equivalent to the Cardinal type. On 64-bit platforms, NativeUInt is equivalent to the UInt64 type.

Signed Integer NativeInt

NativeInt is the platform-dependent signed integer type. The size and internal representation of NativeInt depends on the current platform. On 32-bit platforms, NativeInt is equivalent to the Integer type. On 64-bit platforms, NativeInt is equivalent to the Int64 type.

LongInt and LongWord

LongInt defines the signed integer type and the LongWord defines the unsigned integer type. LongInt and LongWord platform dependent integer types size are changed on each platforms, except for 64-bit Windows that remains unchanged (32-bits).

Size		
	32-bit platforms and 64-bit Windows platforms	**64-bit iOS platforms**
LongInt	32-bits (4 bytes)	64-bits (8 bytes)
LongWord	32-bits (4 bytes)	64-bits (8 bytes)

Note: 32-bit platforms in RAD Studio include 32-bit Windows, OSX32, 32-bit iOS, and Android.

On **64-bit iOS platforms**, if you want to use:

- 32-bits signed integer type, use **Integer** or **FixedInt** instead of LongInt.
- 32-bits unsigned integer type, use **Cardinal** or **FixedUInt** instead of LongWord.

Integer Subrange Types

When you use integer constants to define the minimum and maximum bounds of a subrange type, you define an *integer subrange type*. An integer subrange type represents a subset of the values in an integer type (called the *base type*). The base type is the smallest integer type that contains the specified range (contains both the minimum and maximum bounds).

The internal data format of an integer subrange 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 (FixedInt and LongInt on 32-bit platforms and 64-bit Windows platforms), the variable is stored as a signed double word.
- If both bounds are within the range 0..4294967295 (FixedUInt and LongWord on 32-bit platforms and 64-bit Windows platforms), the variable is stored as an unsigned double word.
- If both bounds are within the range -2^63..2^63-1 (Int64 and LongInt on 64-bit iOS platforms), the variable is stored as a signed quadruple word.
- If both bounds are within the range 0..2^64-1 (UInt64 and LongWord on 64-bit iOS platforms), the variable is stored as an unsigned quadruple word.

  Note: A "word" occupies two bytes.

Character Types

On the 32-bit and 64-bit platforms:

- Char and WideChar are stored as an unsigned word variable, normally using UTF-16 or Unicode encoding.

- AnsiChar type is stored as an unsigned byte. In Delphi 2007 and earlier, **Char** was represented as an **AnsiChar**. The character type used with Short Strings is always AnsiChar and is stored in unsigned byte values.
- The default long string type (string) is now UnicodeString, which is reference counted like an AnsiString, the former default long string type. Compatibility with older code may require the use of the AnsiString type.
- WideString is composed of WideChars like UnicodeString, but is not reference counted.

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 * 2^exponent
```

where the *significand* has a single bit to the left of the binary decimal point (that is, `0 <= significand < 2`).

In the images 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

On the 32-bit and 64-bit platforms, 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^(e-129) * (1.f)
```

If `e = 0`, then `v = 0`.

The **Real48** type cannot store denormals, NaNs, and infinities (Inf). 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

On 32-bit and 64-bit platforms, 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^(e-127) * (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

The **Real** type, in the current implementation, is equivalent to **Double**.

On 32-bit and 64-bit platforms, 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 $v = (-1)^s * 2^{(e-1023)} * (1.f)$
- 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

Extended offers greater precision on 32-bit Intel platform than other real types, but is less portable. Be careful using **Extended** if you are creating data files to share across platforms. Be aware that:

On 32-bit Intel platform, an **Extended** number is represented as 10 bytes (80 bits). An **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 $v = (-1)^s * 2^{(e-16383)} * (i.f)$
- If `e = 32767` and `f = 0`, then $v = (-1)^s * Inf$
- If `e = 32767` and `f <> 0`, then `v` is a NaN

However, on the 64-bit Intel platform and ARM platform, the **Extended** type is an alias for **Double**, which is only 8 bytes. This difference can adversely affect numeric precision in floating-point operations. For more information, see Delphi Considerations for Multi-Device Applications. On MAC OS X systems, the size of **Extended** is 16 bytes in order to be compatible with BCCOSX.

The Comp type

An 8-byte (64-bit) **Comp** number is stored as a signed 64-bit integer.

The Currency type

An 8-byte (64-bit) **Currency** number is stored as a scaled and signed 64-bit integer with the 4 least significant digits implicitly representing 4 decimal places.

Pointer Types

On 32-bit platforms, a pointer type is stored in 4 bytes as a 32-bit address.

On 64-bit platforms, a pointer type is stored in 8 bytes as a 64-bit address.

The pointer value **nil** is stored as zero.

Short String Types

A ShortString 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. The maximum string length is 255 characters plus a length byte (string[255]).

Long String Types

A string variable of type **UnicodeString** or **AnsiString** occupies 4 bytes of memory on 32-bit platforms (and 8 bytes on 64-bit) that contain a pointer to a dynamically allocated string. When a 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 information describing the string. The tables below show the layout of a long-string memory block.

Format of UnicodeString data type (32-bit and 64-bit)

Field	CodePage	ElementSize	ReferenceCount	Length	String Data (ElementSized)	Null Term
Offset	`-12`	`-10`	`-8`	`-4`	`0..(Length - 1)`	`Length * ElementSize`
Contents	16-bit codepage of string data	16-bit element size of string data	32-bit reference-count	Length in characters	Character string of ElementSized data	NULL character

Numbers in the **Offset** row show offsets of fields, describing the string contents, from the string pointer, which points to the **String Data** field (`offset = 0`), containing a block of memory that contains the actual string values.

The NULL character at the end of a string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a string directly to a null-terminated string.

See also "New String Type: UnicodeString."

For string literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of -1. String

constants are treated the same way, the only difference from literals being that they are a pointer to a -1 reference counter block.

When a pointer to a string structure (**source**) is assigned to a string variable (**destination**), the reference counter dictates how this is done. Usually, the reference count is decreased for the destination and increased for the source, as both pointers, source and destination, will point to the same memory block after the assignment.

If the source reference count is -1 (string constant), a new structure is created with a reference count of 1. If the destination is not **nil**, the reference counter is decreased. If it reaches 0, the structure is deallocated from the memory. If the destination is **nil**, no additional actions are taken for it. The destination will then point to the new structure.

```
var
 destination : String;
 source : String;
...
destination := 'qwerty';  // reference count for the newly-created block of
memory (containing the 'qwerty' string) pointed at by the "destination"
variable is now 1
...
source := 'asdfgh'; // reference count for the newly-created block of memory
(containing the 'asdfgh' string) pointed at by the "destination" variable is
now 1
destination := source; // reference count for the memory block containing the
'asdfgh' string is now 2, and since reference count for the block of memory
containing the 'qwerty' string is now 0, the memory block is deallocated.
```

If the source reference count is not -1, it is incremented and the destination will point to it.

```
var
  destination, destination2, destination3: String;
  destination := 'Sample String'; //reference count for the newly-created block
of memory containing 'Sample string' is 1.
  destination2 := destination; //reference count for the block of memory
containing 'Sample string' is now 2.
  destination3 := destination; //reference count for the block of memory
containing 'Sample string' is now 3.
```

> **Note:** No string variable can point to a structure with a reference count of 0. Structures are always deallocated when they reach 0 reference count and cannot be modified when they have -1 reference count.

Wide String Types

On 32-bit platforms, a wide string variable occupies 4 bytes of memory (and 8 bytes on 64-bit) that 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 (32-bit and 64-bit)

Offset	-4	0..(Length - 1)	Length
Contents	32-bit length indicator (in bytes)	Character string	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.

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 platform-dependent integer type or if the program contains code that takes the address of the set.

Static Array Types

On the 32-bit platform, a static-array variable occupies 4 bytes of memory (and 8 bytes on 64-bit) that contain a pointer to the statically allocated array. A static array is stored as a contiguous sequence of elements 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.

Dynamic Array Types

On the 32-bit platform, a dynamic-array variable occupies 4 bytes of memory (and 8 bytes on 64-bit) that 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 (64-bit on Win64) length indicator and a 32-bit reference count. The table below shows the layout of a dynamic-array memory block.

Dynamic array memory layout (32-bit and 64-bit)

Offset 32-bit	`-8`	`-4`	`0..(Length * Size_of_element - 1)`
Offset 64-bit	`-12`	`-8`	`0..(Length * Size_of_element - 1)`
Contents	32-bit reference-count	32-bit or 64-bit on 64-bit platform length indicator (number of elements)	Array elements

Record Types

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, and according to the platform. The alignment is controlled by the type of each field. 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 on which a value of the type must be stored in order to provide the most efficient access. The table below lists the alignments for all data types.

Type alignment masks (32-bit 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 3 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.

Implicit Packing of Fields with a Common Type Specification

Earlier versions of the Delphi compiler, such as Delphi 7 and earlier, implicitly applied **packed** alignment to fields that were declared together, that is, fields that have a common type specification. Newer compilers can reproduce the behavior if you specify the directive {$OLDTYPELAYOUT ON}. This directive byte-aligns (packs) the fields that have a common type specification, even if the declaration does not include the **packed** modifier and the record type is not declared in the `{$A-}` state.

Thus, for example, given the following declaration:

```
{$OLDTYPELAYOUT ON}
type
  TMyRecord = record
    A, B: Extended;
    C: Extended;
  end;
{$OLDTYPELAYOUT OFF}
```

`A` and `B` are packed (aligned on byte boundaries) because the {$OLDTYPELAYOUT ON} directive is specified and because `A` and `B` share the same type specification. However, for the separately declared `C` field, the compiler uses the default behavior and pads the structure with unused bytes to ensure the field 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 is 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

File types are represented as records. Typed files and untyped files occupy 592 bytes on 32-bit platforms and 616 bytes on 64-bit platforms, which are laid out as follows:

```
type
  TFileRec = packed record
    Handle: NativeInt;
    Mode: word;
    Flags: word;
    case Byte of
      0: (RecSize: Cardinal);
      1: (BufSize: Cardinal;
          BufPos: Cardinal;
          BufEnd: Cardinal;
          BufPtr: _PAnsiChr;
          OpenFunc: Pointer;
          InOutFunc: Pointer;
          FlushFunc: Pointer;
          CloseFunc: Pointer;
          UserData: array[1..32] of Byte;
          Name: array[0..259] of WideChar; );
 end;
```

Text files occupy 730 bytes on Win 32 and 754 bytes on Win64, which are laid out as follows:

```
type
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle: NativeInt;
    Mode: word;
    Flags: word;
    BufSize: Cardinal;
    BufPos: Cardinal;
    BufEnd: Cardinal;
    BufPtr: _PAnsiChr;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of WideChar;
    Buffer: TTextBuf; //
    CodePage: Word;
    MBCSLength: ShortInt;
    MBCSBufPos: Byte;
    case Integer of
      0: (MBCSBuffer: array[0..5] of _AnsiChr);
      1: (UTF16Buffer: array[0..2] of WideChar);
  end;
```

`Handle` contains the handle of the file (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.

> **Note:** For using the **UnicodeString** type (the default Delphi string type), the various stream types in the **Classes** unit (**TFileStream**, **TStreamReader**, **TStreamWriter**, and so forth) are more useful, since the older file types have limited Unicode functionality, particularly the old text file type.

Procedural Types

On the 32-bit 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 64-bit platform, a procedure pointer is stored as a 64-bit pointer to the entry point of a procedure or function. A method pointer is stored as a 64-bit pointer to the entry point of a method, followed by a 64-bit pointer to an object.

Class Types

On the 32-bit platforms (Win32, OSX, iOS and Android), a class-type value is stored as a 32-bit pointer to an instance of the class (and as a 64-bit pointer on the 64-bit platform), which is called an *object*. The internal data format of an object resembles that of a record. The fields of the object are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Therefore, the alignment corresponds to the largest alignment of the fields in the *object*. Any fields inherited from an ancestor class are stored before the new fields defined in the descendent class.

On the 32-bit platforms, the first 4-byte field of every object (the first 8-byte field on the 64-bit platform) 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. VMTs are built automatically by the compiler, and are never directly manipulated by a program. Pointers to VMTs, 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. On the 32-bit platforms, at positive offsets, a VMT consists of a list of 32-bit method pointers (64-bit method pointers on the 64-bit platform)--one per user-defined virtual method in the class type--in order of declaration. Each slot contains the address of the

corresponding entry point of the virtual method. 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

Offset Win32, OSX	Offset Win64	Offset iOS/ARM, Android/ARM	Offset iOS/Simulator	Type	Description	Constant in System.pas
-88	-200	-108	-96	Pointer	Pointer to virtual method table (or **nil**)	vmtSelfPtr
-84	-192	-104	-92	Pointer	Pointer to interface table (or **nil**)	vmtIntfTable
-80	-184	-100	-88	Pointer	Pointer to Automation information table (or **nil**)	vmtAutoTable
-76	-176	-96	-84	Pointer	Pointer to instance initialization table (or **nil**)	vmtInitTable
-72	-168	-92	-80	Pointer	Pointer to type information table (or **nil**)	vmtTypeInfo
-68	-160	-88	-76	Pointer	Pointer to field definition table (or **nil**)	vmtFieldTable
-64	-152	-84	-72	Pointer	Pointer to method definition table (or **nil**)	vmtMethodTable
-60	-144	-80	-68	Pointer	Pointer to dynamic method table (or **nil**)	vmtDynamicTable
-56	-136	-76	-64	Pointer	Pointer to short string containing class name	vmtClassName
-52	-128	-72	-60	Cardinal	Instance size in bytes	vmtInstanceSize

-48	-120	-68	-56	Pointer	Pointer to a pointer to ancestor class (or **nil**)	vmtParent
n/a	n/a	-64	-52	Pointer	Entry point of __ObjAddRef method	vmtObjAddRef
n/a	n/a	-60	-48	Pointer	Entry point of __ObjRelease method	vmtObjRelease
-44	-112	-56	-44	Pointer	Entry point of Equals method	vmtEquals
-40	-104	-52	-40	Pointer	Entry point of GetHashCode method	vmtGetHashCode
-36	-96	-48	-36	Pointer	Entry point of ToString method	vmtToString
-32	-88	-44	-32	Pointer	Pointer to entry point of *SafecallException* method (or **nil**)	vmtSafeCallException
-28	-80	-40	-28	Pointer	Entry point of *AfterConstruction* method	vmtAfterConstruction
-24	-72	-36	-24	Pointer	Entry point of *BeforeDestruction* method	vmtBeforeDestruction
-20	-64	-32	-20	Pointer	Entry point of *Dispatch* method	vmtDispatch
-16	-56	-28	-16	Pointer	Entry point of *DefaultHandler* method	vmtDefaultHandler
-12	-48	-24	-12	Pointer	Entry point of *NewInstance* method	vmtNewInstance
-8	-40	-20	-8	Pointer	Entry point of *FreeInstance* method	vmtFreeInstance

-4	-32	-16	-4	Pointer	Entry point of *Destroy* destructor	vmtDestroy
0	0	0	0	Pointer	Entry point of first user-defined virtual method	
4	8	4	4	Pointer	Entry point of second user-defined virtual method	

Class Reference Types

On the 32-bit (Win32, OSX, iOS and Android) platform, a class-reference value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

On the 64-bit (Win64) platform, a class-reference value is stored as a 64-bit pointer to the virtual method table (VMT) of a class.

Variant Types

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 32-bit 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. On the 64-bit platform, a variant is stored as a 24-byte record. The System and System.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 TVarData record contains the following fields:

- The **VType** field of the TVarType type has the Word (16-bit) size. **VType** contains the type code of the variant in the lower 12 bits (the bits defined by the varTypeMask = `$FFF` constant). In addition, the varArray = `$2000` bit may be set to indicate that the variant is an array, and the varByRef (= `$4000`) bit may be set to indicate that the variant contains a reference as opposed to a value.
- The **Reserved1**, **Reserved2**, and **Reserved3** (Word size) fields are unused.

The contents of the remaining 8 bytes (32-bit platform) or 16 bytes (64-bit platform) of a TVarData record depend on the **VType** field as follows:

- 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 the Windows platform, Delphi's Automation support automatically converts varString variants to varOleStr variants before passing them as parameters to external functions.

Program Control (Delphi)

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 **inherited** 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 actions such 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 termination whether 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 **nil**. 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 Index

This section describes the use of the Delphi inline assembler.

Note: *Inline assembly code is supported on the Win32, Win64 and OS X platforms, but is not supported by the Delphi compilers for the iOS device and Android device.*

Topics

Using Inline Assembly Code

The built-in assembler allows you to write assembly code within Delphi programs. 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!).
- Supports the Intel 64 architecture, with some limitations.
- Permits the use of Delphi identifiers, such as constants, types, and variables in assembly statements.
- Provides no macro support, but allows for pure assembly function procedures.

As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions. See the topic 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.

The inline assembler is available on:

- DCC32.EXE, the Delphi Command Line Compiler
- DCC64.EXE, the Delphi 64-bit Command Line Compiler
- DCCOSX.EXE, the Delphi Compiler for OS X

However, inline assembly is not supported by the Delphi compilers for the iOS device and Android device.

Using the asm Statement

The built-in assembler is accessed through **asm** statements, which have the form:

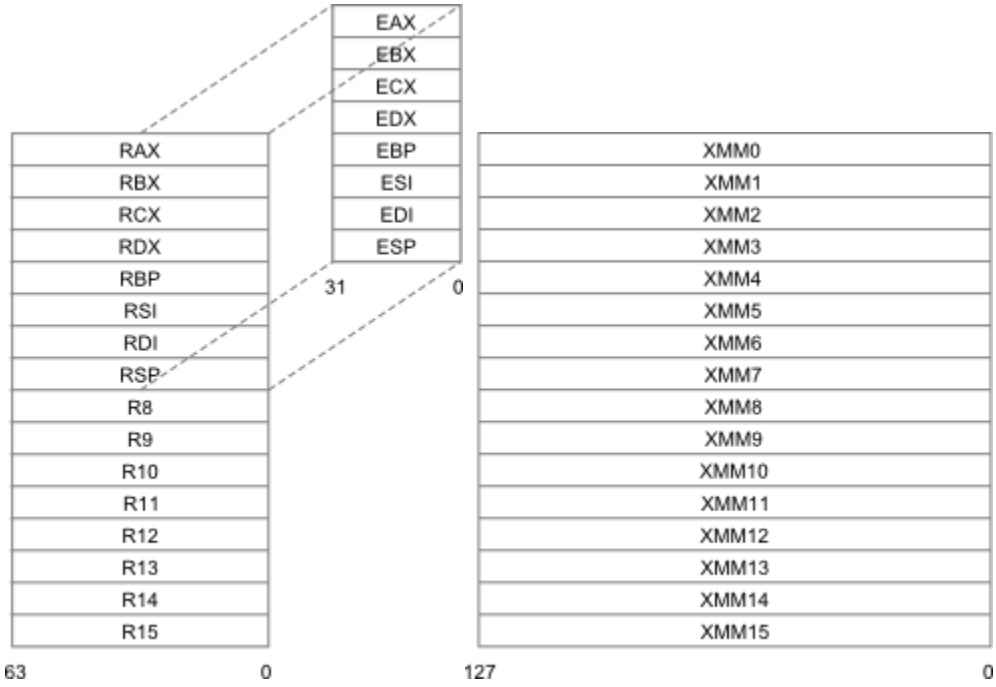
```
asm statementList end
```

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



32-bit

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.

64-bit

In line with the x64 Application Binary Interface (ABI), the contents of the following registers must be preserved and restored within inline assembly functions: R12, R13, R14, R15, RDI, RSI, RBX, RBP, RSP, XMM4, XMM5, XMM6, XMM7, XMM8, XMM8, XMM9, XMM10, XMM11, XMM12, XMM13, XMM14, and XMM15.

The first four parameters to inline assembler functions are passed via RCX, RDX, R8, and R9 respectively, except for floating-point arguments which use XMMO, XMM1, XMM2, XMM3. The math coprocessor is not normally used from x64 code. Registers used for function parameters can be modified freely.

Using Conditional Defines for Cross-Platform Code

For existing functions with inline Assembly code, conditional defines must be used to differentiate between platforms. Functions should have a common function prototype between platforms. Example:

```
function Power10(val: Extended; power: Integer): Extended;
{$IFDEF PUREPASCAL}
begin
  // Pascal implementation here...
end;
{$ELSE !PUREPASCAL}
{$IFDEF CPUX86}
  asm
    // ASM implementation here...
  end;
{$ENDIF CPUX86}
{$ENDIF !PUREPASCAL}
```

Example without `$ELSE`:

```
  {$IFDEF CPUX86}
  asm
    // ...
  end;
  {$ENDIF CPUX86}
  {$IFDEF CPUX64}
  asm
    // ...
  end;
{$ENDIF CPUX64}
```

For more information about the predefined conditionals, see Conditional compilation. Predefined Conditionals.

Assembler Syntax

The following material describes the elements of the assembler syntax.

Statements

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 language by 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:

- IA-32

Pentium family

Pentium Pro and Pentium II

Pentium III

Pentium 4

- Intel 64

In addition, the built-in assembler supports the following instruction set extensions

- Intel SSE (including SSE4.2)
- 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.

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 near jump to the target label.

Jumps to the entry points of procedures and functions are always near.

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.

Directive	Description
DB	Define byte: 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.
DW	Define word: 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.
DD	Define double word: 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.
DQ	Define quad word: 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 does not 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

CPU registers

Category	Identifiers
8-bit CPU registers	AH, AL, BH, BL, CH, CL, DH, DL (general purpose registers);
16-bit CPU registers	AX, BX, CX, DX (general purpose registers); DI, SI, SP, BP (index registers); CS, DS, SS, ES (segment registers); IP (instruction pointer)
32-bit CPU registers	EAX, EBX, ECX, EDX (general purpose registers); EDI, ESI, ESP, EBP (index registers); FS, GS (segment registers); EIP
FPU	ST(0), ..., ST(7)
MMX FPU registers	mm0, ..., mm7
XMM registers	xmm0, ..., xmm7 (..., xmm15 on x64)
Intel 64 registers	RAX, RBX, ...

Data and Operators

Category	Identifiers
Data	BYTE, WORD, DWORD, QWORD, TBYTE
Operators	NOT, AND, OR, XOR; SHL, SHR, MOD; LOW, HIGH; OFFSET, PTR, TYPE
	VMTOFFSET, DMTINDEX
	SMALL, LARGE

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

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants.

Expressions are built from expression elements and operators, and each expression has an associated expression class and expression type.

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'
'Windows'
"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:

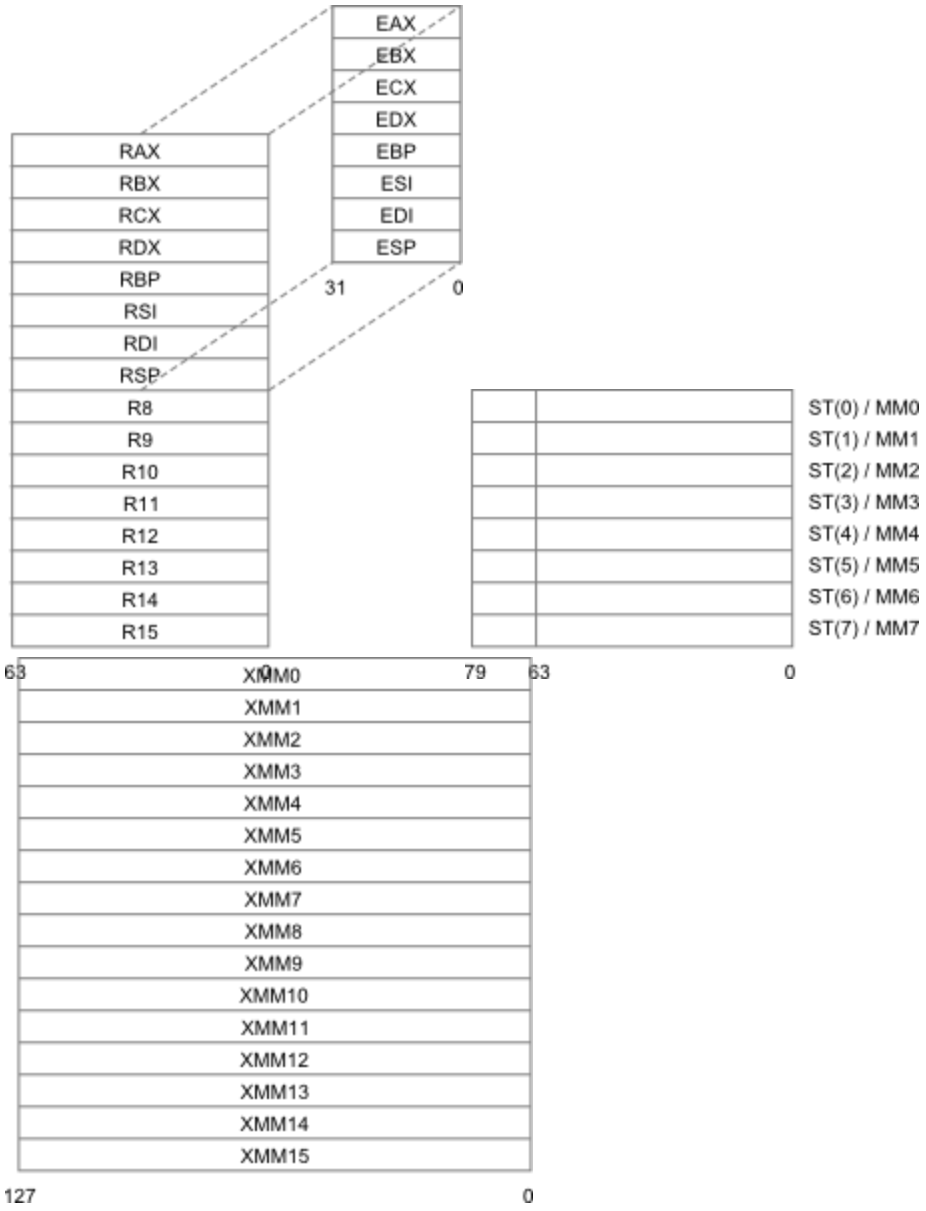
String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers in the inline assembler:
CPU registers

Category	Identifiers
8-bit CPU registers	AH, AL, BH, BL, CH, CL, DH, DL (general purpose registers);
16-bit CPU registers	AX, BX, CX, DX (general purpose registers); DI, SI, SP, BP (index registers); CS, DS, SS, ES (segment registers); IP (instruction pointer)
32-bit CPU registers	EAX, EBX, ECX, EDX (general purpose registers); EDI, ESI, ESP, EBP (index registers); FS, GS (segment registers); EIP
FPU	ST(0), ..., ST(7)
MMX FPU registers	mm0, ..., mm7
XMM registers	xmm0, ..., xmm7 (..., xmm15 on x64)
Intel 64 registers	RAX, RBX, ...

x64 CPU General purpose registers, x86 FPU data registers, and x64 SSE data registers



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 type, or 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 type. [Buffer] 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

You can write complete procedures and functions using inline assembly language code, without including a `begin...end` statement.

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.

32-bit

- 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.

64-bit

- Values 8 bytes or less in size are return in RAX.
- Real values are returned in XMM0.
- Other types are returned by a reference whose pointer value resides in RAX whose memory is allocated by the calling routine.

Intel 64 Specifics (Pseudo-Ops)

x64 functions must be written completely in assembly or Pascal, that is, assembler statements are not supported, only inline assembly functions.

Pseudo-ops have been provided to help manage stack use in x64: .PARAMS, .PUSHNV, .SAVENV and .NOFRAME.

Pseudo-op	Description
`.PARAMS <number>`	Used when calling external functions to setup the register parameter backing store as per the x64 calling convention as this is not normally done by default. When used, a pseudo-variable, @params, is available for passing stack params to called functions. Use @params as a byte array where the first stack parameter will be @params[32], locations 0-31 represent the 4 register parameters.
`.PUSHNV <register>`	Generates code to save and restore the non-volatile general purpose register in the prologue and epilogue.
`.SAVENV <XMM register>`	Same functionality as .PUSHNV for non-volatile XMM registers.
`.NOFRAME`	Forcibly disables the generation of a stack frame as long as there are no local variables declared and the parameter count <= 4. Use **only** for leaf functions.

Stack Unwinding for PC-mapped Exceptions

See PC-Mapped Exceptions#Unwinding Assembly Routines.

Generics Index

Presents an overview of generics, a terminology list, a summary of grammar changes for generics, and details about declaring and using parameterized types, specifying constraints on generics, and using overloads.

Topics

- Overview of Generics
- Terminology for Generics
- Declaring Generics
- Overloads and Type Compatibility in Generics
- Constraints in Generics
- Class Variable in Generics

Overview of Generics

Delphi supports the use of generics.

How Generics Work

The terms **generics** or **generic types** describe the set of things in a platform that can be parameterized by type. The term **generics** can refer to either generic types or generic methods, i.e., generic procedures and generic functions.

Generics are a set of abstraction tools that permit the decoupling of an algorithm (such as a procedure or function) or a data structure (such as a class, interface, or record) from one or more particular types that the algorithm or data structure uses.

A method or data type that uses other types in its definition can be made more general by substituting one or more particular types with type parameters. Then you add those type parameters to a type parameter list in the method or data structure declaration. This is similar to the way that you can make a procedure more general by substituting instances of a literal constant in its body with a parameter name and adding the parameter to the parameter list of the procedure.

For example, a TMyList class that maintains a list of objects (of the TObject type) can be made more reusable and type-safe by substituting uses of TObject with a type parameter name (such as 'T'), and adding the type parameter to the class's type parameter list so that it becomes TMyList<T>.

Particular uses (**instantiations**) of a generic type or method can be made by supplying type arguments to the generic type or method at the point of use. The act of supplying type arguments effectively constructs a new type or method by substituting instances of the type parameter in the generic definition with the corresponding type argument.

For example, the list might be used as TMyList<Double>. This creates a new type, TMyList<Double>, whose definition is identical to TMyList<T> except that all instances of 'T' in the definition are replaced with 'Double'.

It should be noted that generics as an abstraction mechanism duplicates much of the functionality of **polymorphism**, but with different characteristics. Since a new type or method is constructed at instantiation time, you can find type errors sooner, at compile time rather than at run time. This also increases the scope for optimization, but with a trade-off - each instantiation increases the memory usage of the final running application, possibly resulting in lower performance.

Code Examples

For example, TSIPair is a class holding two data types, String and Integer:

```
type
  TSIPair = class
  private
    FKey: String;
    FValue: Integer;
  public
    function GetKey: String;
    procedure SetKey(Key: String);
    function GetValue: Integer;
    procedure SetValue(Value: Integer);
      property Key: TKey read GetKey write SetKey;
      property Value: TValue read GetValue write SetValue;
  end;
```

To make a class independent of data type, replace the data type with a type parameter:

```
type
  TPair<TKey,TValue> = class   // declares TPair type with two type parameters

  private
    FKey: TKey;
    FValue: TValue;
  public
    function GetKey: TKey;
    procedure SetKey(Key: TKey);
    function GetValue: TValue;
    procedure SetValue(Value: TValue);
    property Key: TKey read GetKey write SetKey;
    property Value: TValue read GetValue write SetValue;
  end;

type
  TSIPair = TPair<String,Integer>; // declares instantiated type
  TSSPair = TPair<String,String>;  // declares with other data types
  TISPair = TPair<Integer,String>;
  TIIPair = TPair<Integer,Integer>;
  TSXPair = TPair<String,TXMLNode>;
```

Platform Requirements and Differences

Generics are supported by the Delphi compilers.

Run-time type identification

In Win32, generics and methods do not have run-time type information (RTTI), but instantiated types do have RTTI. An **instantiated type** is the combination of a generic with a set of parameters. The RTTI for methods of a class is a subset of the RTTI for that class as a whole. If a non-generic class has a generic method, that method will not be included in the RTTI generated for the class because generics are instantiated at compile time, not at run time.

Interface GUID

In Win32, an instantiated interface type does not have an interface GUID.

Parameterized method in interface

A parameterized method (method declared with type parameters) cannot be declared in an interface.

Instantiation timing

Generic types are instantiated at compile time and emitted into executables and relocatable files. Instance variables of a generic type are instantiated at run time for classes and at compile time for generic records. The RTTI for generic classes is only generated when the classes are instantiated. RTTI for instantiated classes follows just as for non-generic classes. If the generic class has a generic method, then the instantiated generic class will not have RTTI for that generic method.

Dynamic instantiation

Dynamic instantiation at run time is not supported.

Interface constraints

The Win32 interface is not a "light" interface. This means all type parameters with interface constraints always support the COM IUnknown methods **_AddRef**, **_Release**, and **QueryInterface** or inherit from **TInterfacedObject**. Record types cannot specify an interface constraint parameter.

Terminology for Generics

Terminology used to describe generics is defined in this section.

Type generic	A type declaration that requires type parameters to be supplied in order to form an actual type.
	`List<Item>` is a **type generic** (or **generic**) in the following example:
	`type`
	`  List<Item> = class`
	`  ...`
	`  end;`

Generic	Same as **Type generic**.

Type parameter	A parameter declared in a generic declaration or a method header in order to use as a type for another declaration inside its generic declaration or the method body. It will be bound to a particular type argument. `Item` is a type parameter in the following example:
	`type`
	`  List<Item> = class`
	`  ...`
	`  end;`

Type argument and **Type identifier**	A particular type used with a type identifier in order to make an instantiated type. Using the previous example, `List<Integer>` is the instantiated type (instantiated generic), `List` is the type identifier, and `Integer` is the type argument.
Instantiated type	The combination of a generic with a set of parameters.
Constructed type	Same as **instantiated type**.
Closed constructed type	A constructed type having all its parameters resolved to actual types. `List<Integer>` is closed because `Integer` is an actual type.
Open constructed type	A constructed type having at least one parameter that is a type parameter. If `T` is a type parameter of a containing class, `List<T>` is an open constructed type.
Instantiation	The compiler generates real instruction code for methods defined in generics and real virtual method table for a closed constructed type. This process is required before emitting a Delphi compiled unit file (.dcu) or object file (.obj) for Win32.

Declaring Generics

The declaration of a generic is similar to the declaration of a regular class, record, or interface type. The difference is that a list of one or more type parameters placed between angle brackets (< and >) follows the type identifier in the declaration of a generic.

A type parameter can be used as a typical type identifier inside a container type declaration and method body.

For example:

```
type
  TPair<TKey,TValue> = class   // TKey and TValue are type parameters
    FKey: TKey;
    FValue: TValue;
    function GetValue: TValue;
  end;

function TPair<TKey,TValue>.GetValue: TValue;
begin
  Result := FValue;
end;
```

> **Note:** You must call the default constructor and initialize the class fields before calling the **GetValue** method.

Type Argument

Generic types are instantiated by providing type arguments. In Delphi, you can use any type as a type argument except for the following: a static array, a short string, or a record type that (recursively) contains a field of one or more of these two types.

```
type
  TFoo<T> = class
    FData: T;
  end;
var
  F: TFoo<Integer>; // 'Integer' is the type argument of TFoo<T>
begin
  ...
end.
```

Nested Types

A nested type within a generic is itself a generic.

```
type
  TFoo<T> = class
  type
    TBar = class
      X: Integer;
      // ...
    end;
  end;

  // ...
  TBaz = class
  type
    TQux<T> = class
      X: Integer;
      // ...
    end;
    // ...
  end;
```

To access the TBar nested type, you must specify a construction of the TFoo type first:

```
var
  N: TFoo<Double>.TBar;
```

A generic can also be declared within a regular class as a nested type:

```
type
  TOuter = class
  type
    TData<T> = class
      FFoo1: TFoo<Integer>;          // declared with closed constructed type
      FFoo2: TFoo<T>;                // declared with open constructed type
      FFooBar1: TFoo<Integer>.TBar; // declared with closed constructed type
      FFooBar2: TFoo<T>.TBar;        // declared with open constructed type
      FBazQux1: TBaz.TQux<Integer>; // declared with closed constructed type
      FBazQux2: TBaz.TQux<T>;        // declared with open constructed type
      ...
    end;
  var
    FIntegerData: TData<Integer>;
    FStringData: TData<String>;
  end;
```

Base Types

The base type of a parameterized class or interface type might be an actual type or a constructed type. The base type might not be a type parameter alone.

```
type
  TFoo1<T> = class(TBar)            // Actual type
  end;

  TFoo2<T> =  class(TBar2<T>)       // Open constructed type
  end;

  TFoo3<T> = class(TBar3<Integer>)  // Closed constructed type
  end;
```

If TFoo2<String> is instantiated, an ancestor class becomes TBar2<String>, and TBar2<String> is automatically instantiated.

Class, Interface, and Record Types

Class, interface, record, and array types can be declared with type parameters.

For example:

```
type
  TRecord<T> = record
    FData: T;
  end;

type
  IAncestor<T> = interface
    function GetRecord: TRecord<T>;
  end;

  IFoo<T> = interface(IAncestor<T>)
    procedure AMethod(Param: T);
  end;

type
  TFoo<T> = class(TObject, IFoo<T>)
    FField: TRecord<T>;
    procedure AMethod(Param: T);
    function GetRecord: TRecord<T>;
  end;

type
  anArray<T>= array of T;
  IntArray= anArray<integer>;
```

Procedural Types

The procedure type and the method pointer can be declared with type parameters. Parameter types and result types can also use type parameters.

For example:

```
type
  TMyProc<T> = procedure(Param: T);
  TMyProc2<Y> = procedure(Param1, Param2: Y) of object;
type
  TFoo = class
    procedure Test;
    procedure MyProc(X, Y: Integer);
  end;

procedure Sample(Param: Integer);
begin
  Writeln(Param);
end;

procedure TFoo.MyProc(X, Y: Integer);
begin
  Writeln('X:', X, ', Y:', Y);
end;

procedure TFoo.Test;
var
  X: TMyProc<Integer>;
  Y: TMyProc2<Integer>;
begin
  X := Sample;
  X(10);
  Y := MyProc;
  Y(20, 30);
end;

var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  F.Free;
end.
```

Parameterized Methods

Methods can be declared with type parameters. Parameter types and result types can use type parameters. However, constructors and destructors cannot have type parameters, and neither can virtual, dynamic, or message methods. Parameterized methods are similar to overloaded methods.

There are two ways to instantiate a method:

- Explicitly specifying type argument
- Automatically inferring from the type argument

For example:

```
type
  TFoo = class
    procedure Test;
    procedure CompareAndPrintResult<T>(X, Y: T);
  end;

procedure TFoo.CompareAndPrintResult<T>(X, Y: T);
var
  Comparer : IComparer<T>;
begin
  Comparer := TComparer<T>.Default;
  if Comparer.Compare(X, Y) = 0 then
    WriteLn('Both members compare as equal')
  else
    WriteLn('Members do not compare as equal');
end;

procedure TFoo.Test;
begin
  CompareAndPrintResult<String>('Hello', 'World');
  CompareAndPrintResult('Hello', 'Hello');
  CompareAndPrintResult<Integer>(20, 20);
  CompareAndPrintResult(10, 20);
end;

var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  ReadLn;
  F.Free;
end.
```

Scope of Type Parameters

The scope of a type parameter covers the type declaration and the bodies of all its members, but does not include descendent types.

For example:

```
type
  TFoo<T> = class
    X: T;
  end;

  TBar<S> = class(TFoo<S>)
    Y: T;  // error!  unknown identifier "T"
  end;

var
  F: TFoo<Integer>;
begin
  F.T  // error! unknown identifier "T"
end.
```

Overloads and Type Compatibility in Generics

Overloads

Generic methods can participate in overloading alongside non-generic methods by using the 'overload' directive. If overload selection between a generic method and a non-generic method would otherwise be ambiguous, the compiler selects the non-generic method.

For example:

```
type
  TFoo = class
    procedure Proc<T>(A: T); overload;
    procedure Proc(A: String); overload;
    procedure Test;
  end;

procedure TFoo.Test;
begin
  Proc('Hello'); // calls Proc(A: String);
  Proc<String>('Hello'); // calls Proc<T>(A: T);
end;
```

Type Compatibility

Two non-instantiated generics are considered assignment compatible only if they are identical or are aliases to a common type.

Two instantiated generics are considered assignment compatible if the base types are identical (or are aliases to a common type) and the type arguments are identical.

> **Note:** Generics in Delphi are unlike templates in C++ or generic types in C#. Most notably, a **type parameter** cannot be constrained to a specific simple type, such as Integer, Double, String, and so forth. In cases where different types need to be expressed (including simple types), consider using **overloaded functions** for each needed type or use a **record type** such as TValue from the System.Rtti unit, which offers operators and methods for storing and querying these data types.

Constraints in Generics

Constraints can be associated with a type parameter of a generic. Constraints declare items that must be supported by any particular type passed to that parameter in a construction of the generic type.

Specifying Generics with Constraints

Constraint items include:

- Zero, one, or multiple interface types
- Zero or one class type
- The reserved word "constructor", "class", or "record"

You can specify both "constructor" and "class" for a constraint. However, "record" cannot be combined with other reserved words. Multiple constraints act as an additive union ("AND" logic).

The examples given here show only class types, although constraints apply to all forms of generics.

Declaring Constraints

Constraints are declared in a fashion that resembles type declarations in regular parameter lists:

```
type
  TFoo<T: ISerializable> = class
    FField: T;
  end;
```

In the example declaration given here, the 'T' type parameter indicates that it must support the ISerializable interface. In a type construction like TFoo<TMyClass>, the compiler checks at compile time to ensure that TMyClass actually implements ISerializable.

Multiple Type Parameters

When you specify constraints, you separate multiple type parameters by semicolons, as you do with a parameter list declaration:

```
type
   TFoo<T: ISerializable; V: IComparable>
```

Like parameter declarations, multiple type parameters can be grouped together in a comma list to bind to the same constraints:

```
type
   TFoo<S, U: ISerializable> ...
```

In the example above, S and U are both bound to the ISerializable constraint.

Multiple Constraints

Multiple constraints can be applied to a single type parameters as a comma list following the colon:

```
type
   TFoo<T: ISerializable, ICloneable; V: IComparable> ...
```

Constrained type parameters can be mixed with "free" type parameters. For example, all the following are valid:

```
type
   TFoo<T; C: IComparable> ...
   TBar<T, V> ...
   TTest<S: ISerializable; V> ...
   // T and V are free, but C and S are constrained
```

Types of Constraints

Interface Type Constraints

A type parameter constraint may contain zero, one, or a comma separated list of multiple interface types.

A type parameter constrained by an interface type means that the compiler will verify at compile time that a concrete type passed as an argument to a type construction implements the specified interface type(s).

For example:

```
type
  TFoo<T: ICloneable> ...

  TTest1 = class(TObject, ICloneable)
     ...
  end;

  TError = class
  end;

var
  X: TFoo<TTest1>;  //  TTest1 is checked for ICloneable support here
                    //  at compile time
  Y: TFoo<TError>;  //  exp: syntax error here - TError does not support
                    //  ICloneable
```

Class Type Constraints

A type parameter may be constrained by zero or one class type. As with interface type constraints, this declaration means that the compiler will require any concrete type passed as an argument to the constrained type param to be assignment compatible with the constraint class.

Compatibility of class types follows the normal rules of OOP type compatibilty - descendent types can be passed where their ancestor types are required.

Constructor Constraints

A type parameter may be constrained by zero or one instance of the reserved word "constructor". This means that the actual argument type must be a class that defines a default constructor (a public parameterless constructor), so that methods within the generic type may construct instances of the argument type using the default constructor of the argument type, without knowing anything about the argument type itself (no minimum base type requirements).

In a constraint declaration, you can mix "constructor" in any order with interface or class type constraints.

Class Constraint

A type parameter may be constrained by zero or one instance of the reserved word "class". This means that the actual type must be a class type.

Record Constraint

A type parameter may be constrained by zero or one instance of the reserved word "record". This means that the actual type must be a value type (not a reference type). A "record" constraint cannot be combined with a "class" or "constructor" constraint.

Type Inferencing

When using a field or variable of a constrained type parameter, it is not necessary in many cases to typecast in order to treat the field or variable as one of the constrained types. The compiler can infer which type you're referring to by looking at the method name and by performing a variation of overload resolution over the union of the methods sharing the same name across all the constraints on that type.

For example:

```
type
  TFoo<T: ISerializable, ICloneable> = class
    FData: T;
    procedure Test;
  end;

procedure TFoo<T>.Test;
begin
  FData.Clone;
end;
```

The compiler looks for "Clone" methods in ISerializable and ICloneable, since FData is of type T, which is guaranteed to support both those interfaces. If both interfaces implement "Clone" with the same parameter list, the compiler issues an ambiguous method call error and require you to typecast to one or the other interface to disambiguate the context.

Class Variable in Generics

The class variable defined in a generic type is instantiated in each instantiated type identified by the type parameters.

The following code shows that TFoo<Integer>.FCount and TFoo<String>.FCount are instantiated only once, and these are two different variables:

```
{$APPTYPE CONSOLE}
type
  TFoo<T> = class
    class var FCount: Integer;
    constructor Create;
  end;
  constructor TFoo<T>.Create;
begin
  inherited Create;
  Inc(FCount);
end;

procedure Test;
var
  FI: TFoo<Integer>;
begin
  FI := TFoo<Integer>.Create;
  FI.Free;
end;

var
  FI: TFoo<Integer>;
  FS: TFoo<String>;

begin
  FI := TFoo<Integer>.Create;
  FI.Free;
  FS := TFoo<String>.Create;
  FS.Free;
  Test;
  Writeln(TFoo<Integer>.FCount); // outputs 2
  Writeln(TFoo<String>.FCount);  // outputs 1
end.
```

Attributes and RTTI

Introduces the concept of attributes, their general use case, and some of their basic restrictions in the **Delphi** language.

> **Note:** Delphi attributes are not supported in C++Builder. For information about RTTI in C++Builder, see Delphi RTTI and C++Builder.

Introduction

Attributes are a language feature in Delphi that allows annotating types and type members with special objects that carry additional information. This information can be queried at run time. Attributes extend the normal *Object-Oriented* model with *Aspect-Oriented* elements.

In general, attributes are useful when building general purpose frameworks that analyze structured types such as objects or records at run time and introduce new behavior based on additional information supplied by the annotated attributes.

Attributes and RTTI

Attributes do not modify the behavior of types or members by themselves. The consumer code must specifically query for their existence and take appropriate actions when this is required. To be able to attach an attribute to an entity in the compiled binary, first you need to have RTTI information emitted for that entity. This means that types that explicitly disable RTTI information are not eligible for attribute annotation. For example, in the following code, *SomeCustomAttribute* will not be emitted to the compiled binary, because the RTTI information is specifically disabled for the *TDerivedObject* class.

```
type
  {$RTTI EXPLICIT METHODS([]) PROPERTIES([]) FIELDS([])}
  TDerivedObject = class(TObject)
    [SomeCustomAttribute()]
    procedure Shoot;
  end;
```

Topics

- Declaring Custom Attributes (RTTI)
- Annotating Types and Type Members
- Extracting Attributes at Run Time
- Using Virtual Method Interceptors

Declaring Custom Attributes (RTTI)

This topic describes the basic methods used to create custom attributes, the proper design decisions for attribute classes, and the general use cases.

Declaring an Attribute

An attribute is a simple class type. To declare your own custom attribute, you must derive it from a special predefined class: System.TCustomAttribute:

```
type
    MyCustomAttribute = class(TCustomAttribute)
    end;
```

`MyCustomAttribute` can then be used to annotate any type or any member of a type (such as class, record, or interface):

```
type
   [MyCustomAttribute]
   TSpecialInteger = type integer;

   TSomeClass = class
        [MyCustomAttribute]
        procedure Work;
   end;
```

Note that the declared attribute class must not be declared as **class abstract** and should not contain any abstract methods. Even though the compiler allows you to use these attributes for annotation, the built binary will not include them in the emitted RTTI information.

Attribute Names that End with 'Attribute' are Implicitly Shortened

Suppose you define two `TCustomAttribute` subclasses with the same name prefix, but one has 'Attribute' as a suffix, such as:

- `MyCustom`
- `MyCustomAttribute`

The class with the 'Attribute' suffix (`MyCustomAttribute`) is always used, and the class with the shorter name (`MyCustom`) becomes inaccessible.

The following code snippet demonstrates this issue. One might expect the `TCustomAttribute` subclass, `Test`, to be applied but because of implicit name shortening, `TestAttribute` will actually be applied where either `[Test]` or `[TestAttribute]` are used.

```
type
  // To check ambigious names
  TestAttribute = class(TCustomAttribute)
  end;

  // Becomes unaccessible
  Test = class(TCustomAttribute)
  end;

  [Test] // Resolves to TestAttribute at run time
  TAmbigiousClass = class
  end;
```

Constructors in Attributes

Normally, an attribute is designed to carry some additional information that can be queried at run time. To allow specifying custom information for the attribute class, you must declare constructors for it:

```
type
    AttributeWithConstructor = class(TCustomAttribute)
    public
        constructor Create(const ASomeText: String);
   end;
```

which can then be used as follows:

```
 type
   [AttributeWithConstructor('Added text value!')]
   TRecord = record
        FField: Integer;
   end;
```

The method resolution works for attributes as well, which means that you can define overloaded constructors in the custom-defined attribute. Declare only constructors that accept constant values and not `out` or `var` ones. This comes out of a basic restriction in how attributes work and is discussed in more detail in Annotating Types and Type Members.

Annotating Types and Type Members

This topic describes the syntax and rules appropriate when annotating a type or a member with an attribute.

General Syntax

To annotate a Delphi type or a member, such as a class or a class member, you must precede the declaration of that type by the name of the attribute class between brackets:

```
[CustomAttribute]
TMyClass = class;
```

If the name of the attribute class ends in "Attribute", you can also omit the "Attribute" suffix:

```
[Custom]
procedure DoSomething;
```

Having a set of parenthesis after the attribute class name is also a valid syntax:

```
[Custom()]
TMyRecord = record;
```

Some attributes accept parameters. To pass arguments to your attribute, use the same syntax as you use for method calls:

```
[Custom(Argument1, Argument2, …)]
TSimpleType = set of (stOne, stTwo, stThree);
```

To annotate a single type with several attributes, you can either use several sets of brackets:

```
[Custom1]
[Custom2(MyArgument)]
FString: String;
```

Or use comma-separated attributes between a single set of brackets:

```
[Custom1, Custom2(MyArgument)]
function IsReady: Boolean;
```

You Can Only Use Constant Expressions as Attribute Parameters

An attribute that is annotated to a type or a member is inserted into the RTTI information block in the generated binary. The emitted information includes:

- The class type of the attribute.
- The pointer to the selected constructor.
- A list of constants that are later passed to the attribute constructor.

The values passed to the constructor of the attribute must be constant expressions. Because those values must be embedded directly into the resulting binary, it is impossible to pass an expression that requires run-time evaluation. This

raises a few limitations to the information that can be passed to the attribute at compile time:

- You can only use constant expressions, including sets, strings, and ordinal expressions.
- You can use TypeInfo() to pass type information because the RTTI block addresses are known at compile time.
- You can use class references because the metaclass addresses are known at compile time.
- You cannot use out or var parameters because they require run-time evaluation of addresses of passed parameters.
- You cannot use Addr() or @.

The following code exemplifies the case in which the compiler does not compile the annotation:

```
var
    a, b: Integer;
type
    [SomeAttribute(a + b)]
    TSomeType = record
    // …
    end;
```

In the previous example, the constructor of `SomeAttribute` requires an integer value. The passed expression requires a run-time evaluation of `a + b`. The compiler emits a compile-time error because it expects a constant expression.

The code below shows an accepted expression:

```
const
    a = 10;
    b = 20;
type
    [SomeAttribute(a + b)]
    TSomeType = record
    // …
    end;
```

The values of `a` and `b` are known at compile time; thus, the constant expression is evaluated directly.

Extracting Attributes at Run Time

Provides information about run-time aspects of attributes--how to extract them and how to make custom decisions based on their informational value.

Attribute Instantiation

Annotation (as discussed in Annotating Types and Type Members) is a simple method of attaching an attribute to a type or a member. The information included into the compiled binary only includes the class of the attribute, the pointer to the selected constructor, and the list of constants that are passed to the attribute's constructor at **instantiation time**.

The actual instantiation of attributes happens when the consumer code queries for them in a given type or type member. This means that instances of attribute classes are not created automatically, but rather when the program explicitly searches for them. There is no guaranteed order in which attributes are instantiated, nor it is known how many instances are created. A program should not depend on such consequences.

Consider the following attribute declaration:

```
type
  TSpecialAttribute = class(TCustomAttribute)
  public
    FValue: String;

    constructor Create(const AValue: String);
  end;

constructor TSpecialAttribute.Create(const AValue: String);
begin
  FValue := AValue;
end;
```

The *TSpecialAttribute* is then used as annotation in the following example:

```
type
  [TSpecialAttribute('Hello World!')]
  TSomeType = record
  ...
  end;
```

To extract the attribute from the **TSomeType** type, the user code must employ the functionality exposed by the System.Rtti unit. The following example demonstrates the extraction code:

```
var
  LContext: TRttiContext;
  LType: TRttiType;
  LAttr: TCustomAttribute;
begin
  { Create a new Rtti context }
  LContext := TRttiContext.Create

  { Extract type information for TSomeType type }
  LType := LContext.GetType(TypeInfo(TSomeType));

  { Search for the custom attribute and do some custom processing }
  for LAttr in LType.GetAttributes() do
    if LAttr is TSpecialAttribute then
      Writeln(TSpecialAttribute(LAttr).FValue);

  { Destroy the context }
  LContext.Free;
end;
```

As seen in the example above, the user must specifically write code to query for attributes annotated to a type. The actual attribute instances are created in the TRttiType.GetAttributes method. Note that the example does not destroy the instances; the TRttiContext frees all resources afterward.

Exceptions

Because the actual instantiation of attributes is performed in the user code, one must be aware of the possible exceptions that may occur in the attributes' constructors. The general recommendation is to use a **try** .. **except** clause surrounding the code that queries for attributes.

To exemplify the problem, the attribute constructor in the original example is changed to look like this:

```
constructor TSpecialAttribute.Create(const AValue: String);
begin
  if AValue = '' then
    raise EArgumentException.Create('Expected a non-null string');

  FValue := AValue;
end;
```

and the annotation for **TSomeType** is changed to pass an empty string to the attribute constructor:

```
type
  [TSpecialAttribute('')]
  TSomeType = record
  ...
  end;
```

In this case, the code that queries for the attributes of type **TSomeType** will fail with an **EArgumentException** exception, which is raised by the instantiating attribute. The recommendation is to change the query code to use the **try** .. **except** clause:

```
{ Search for the custom attribute and do some custom processing }
try
  for LAttr in LType.GetAttributes() do
    if LAttr is TSpecialAttribute then
      Writeln(TSpecialAttribute(LAttr).FValue);
except
  { ... Do something here ... }
end;
```

Using Virtual Method Interceptors

Delphi has a new type in Rtti.pas called System.Rtti.TVirtualMethodInterceptor. Essentially, this type creates a derived metaclass dynamically at run time that overrides every virtual method in the ancestor, by creating a new virtual method table and populating it with stubs that intercept calls and arguments. When the metaclass reference for any instance of the "ancestor" is replaced with this new metaclass, the user can then intercept virtual function calls, change arguments on the fly, change the return value, intercept and suppress exceptions or raise new exceptions, or entirely replace calling the underlying method.

In concept, this is somewhat similar to dynamic proxies from .NET and Java. It is like being able to derive from a class at run time, override methods (but not add new instance fields), and then change the run-time type of an instance to this new derived class.

For more information, see Barry Kelly's blog at http://blog.barrkel.com/2010/09/virtual-method-interception.html

Compiler Attributes

Some special attributes trigger certain features of Delphi compilers.

Ref

The `Ref` attribute is used to qualify constant function parameters so that they are passed by reference (not by value) to the function. For more information, see **Constant Parameters**.

Unsafe

Tag the `Result` of a function as `Unsafe` to make the compiler treat the function result as "unsafe", which disables ARC management of the object. For more information, see **The Unsafe Attribute**.

Volatile

The `volatile` attribute is used to mark fields that are subject to change by different threads, so that code generation does not optimize copying the value in a register or another temporary memory location.

You can use the `volatile` attribute to mark the following declarations:

- Variables (global and local)
- Parameters
- Fields of a record or a class.

You cannot use the `volatile` attribute to mark the following declarations:

- Type
- Procedures, Functions or Methods
- Expressions

```
type
    TMyClass = class
    private
        [volatile] FMyVariable: TMyType;
    end;
```

Weak

The `weak` attribute is used to mark a declaration as a weak reference. For more information, see **Weak References**.

Writing C++-friendly Delphi Code

C++ can consume Delphi code. The Delphi command-line compiler uses the following switches to generate the files that C++ needs to process Delphi code:

- The `-JL` switch generates .lib, .bpi, .bpl and .obj files from a .dpk file, and header files for all units in the package.
- The `-JPHNE` switch does the same from a `.pas` unit.

However, not all Delphi features are C++-friendly. This topic lists the DOs and DON'Ts for Delphi run-time code that you want to consume from C++.

DOs

Redeclaring All Inhereted Contructors

Unlike Delphi, C++ does not inherit constructors. For example, the following is incorrect:

```
class A
{
  public:
  A(int x) {}
};

class B: public A
{
};

int main(void)
{
  B *b = new B(5);  // Error
  delete b;
}
```

The header file generation logic of the Delphi compiler is aware of this language difference and adds the missing inherited constructors to each derived class. However, these constructors also initialize member variables of the class. This causes problems if a base class invokes a virtual method that already initialized one of these member variables to a non-default value. It is particularly important to redeclare inherited constructors if the base constructor can initialize a member of a delphireturn type in the class.

Ensuring Distinct Signature for Each Constructor in a Hierarchy

C++ does not support named constructors. For this reason overloaded constructors must not have identical nor similar parameters. For example, the following code does not work for C++ consumption:

```
MyPoint = class
public
  constructor Polar(Radius, Angle: Single);
  constructor Rect(X, Y: Single);
```

This example results in the following C++ code that arises compilation errors associated with duplicated constructors:

```
class PASCALIMPLEMENTATION MyPoint : public System::TObject
{
  public:
    __fastcall MyPoint(float Radius, float Angle);
    __fastcall MyPoint(float X, float Y);
};
```

You can workaround this issue in different ways:

- Add a dummy parameter with a default value to one of the constructors. The header file generation logic intentionally leaves out the default value on the constructor so that the two constructors are distinct in C++:

```
MyPoint = class
public
  constructor Polar(Radius, Angle: Single);
  constructor Rect(X, Y: Single; Dummy: Integer = 0);
```

```
class PASCALIMPLEMENTATION MyPoint : public System::TObject
{
  public:
    __fastcall MyPoint(float Radius, float Angle);
    __fastcall MyPoint(float X, float Y, int Dummy);
};
```

- Use the Named Constructor Idiom. This technique declares class static factory members instead of named constructors when constructors are overloaded with identical or similar parameters. This is particularly relevant for the Delphi record type. The following example depicts a solution based on this technique:

```
class MyPoint {
public:
  static MyPoint Rect(float X, float Y);  // Rectangular coordinates
  static MyPoint Polar(float Radius, float Angle);  // Polar coordinates
private:
  MyPoint(float X, float Y);  // Rectangular coordinates
  float X_, Y_;
};

inline MyPoint::MyPoint(float X, float Y)
  : X_(X), Y_(Y) { }

inline MyPoint MyPoint::Rect(float X, float Y)
{  return MyPoint(X, Y); }

inline MyPoint MyPoint::Polar(float Radius, float Angle)
{  return Point(Radius*std::cos(Angle), Radius*std::sin(Angle)); }
```

DON'Ts

Overloading Index Properties

Delphi allows overloading index properties, such as:

```
TTest = class
  function  GetPropI(Index: Integer): Longint; overload;
  procedure SetProp(Index: Integer; Value: Longint); overload;
  function  GetPropS(Index: Integer): String; overload;
  procedure SetProp(Index: Integer; Value: String); overload;
public
  property Props[Index: Integer] : Longint read GetPropI write SetProp;
  property Props[Index: Integer] : String read GetPropS write SetProp; default;
end;
```

However, the resulting interface in the header file does not work in C++, since each property of a class must be unique.

Calling Virtual Mehtods from Constructors

This is related to Redeclaring All Inhereted Contructors. For Delphi-style classes, the vtable of the most-derived class is set when the base constructors is invoked. This allows the virtual mechanism to work from constructors. However, this implies a strange behavior in a C++ environment, such as a virtual method of a class that is invoked before the constructor of the class executes; or the constructor of a class that undoes the initialization of a member that was performed from a base constructor.

Using Generics in Aliases

C++ can use a Delphi alias to an instantiated template type. However, C++ cannot use a Delphi alias with dependent types. The following code illustrates this fact:

```
type
  GoodArray = TArray<Integer>;
  BadArray<T> = array of T;
```

`GoodArray` is a concrete type that C++ can use. In contrast, `BadArray` contains a dependent type, thus C++ cannot use it.

Using Generics in Closures

RTTI generated for published events allows the IDE to generate event handlers. The logic in the IDE is unable to process RTTI generated for Generics when C++ event handlers are generated. Thus it is recommended that you avoid using Generics in closures.

Using Records with Constructors

In Delphi, a variant record is equivalent to a C++ union. Records with constructors cannot be in a variant record. The C++ rule is actually more generic: a type with a user-defined constructor, destructor, or assignment cannot be a member of a union. The following code illustrates a case that does not work for C++:

```
type
  TPointD = record
    X: Double;
    Y: Double;
  public
    constructor Create(const X, Y: Double);
  end;

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

The resulting C++ code triggers compiler errors:

```
struct DECLSPEC_DRECORD TRectD
{
  #pragma pack(push,1)
   union
    {
      struct
      {
        TPointD TopLeft;     // Error
        TPointD BottomRight; // Error
      };
      struct
      {
        double Left;
        double Top;
        double Right;
        double Bottom;
      };

    };
  #pragma pack(pop)
};
```

Using Non-Empty Default String Parameters

Non-empty default string parameters generate the following warning:

```
W8058 Cannot create pre-compiled header: initialized data in header
```

Note that this issue only affects previous-generation C++ compilers (BCC32 and BCCOSX), it does not affect Clang-enhanced C++ compilers.

Creative Commons License Deed

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

This is a human-readable summary of (and not a substitute for) the license.

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Creative Commons Legal Code

Attribution-ShareAlike 4.0 International

Official translations of this license are available in other languages.

Creative Commons Corporation (“Creative Commons”) is not a law firm and does not provide legal services or legal advice. Distribution of Creative Commons public licenses does not create a lawyer-client or other relationship. Creative Commons makes its licenses and related information available on an “as-is” basis. Creative Commons gives no warranties regarding its licenses, any material licensed under their terms and conditions, or any related information. Creative Commons disclaims all liability for damages resulting from their use to the fullest extent possible.

Using Creative Commons Public Licenses

Creative Commons public licenses provide a standard set of terms and conditions that creators and other rights holders may use to share original works of authorship and other material subject to copyright and certain other rights specified in the public license below. The following considerations are for informational purposes only, are not exhaustive, and do not form part of our licenses.

Considerations for licensors: Our public licenses are intended for use by those authorized to give the public permission to use material in ways otherwise restricted by copyright and certain other rights. Our licenses are irrevocable. Licensors should read and understand the terms and conditions of the license they choose before applying it. Licensors should also secure all rights necessary before applying our licenses so that the public can reuse the material as expected. Licensors should clearly mark any material not subject to the license. This includes other CC-licensed material, or material used under an exception or limitation to copyright.

Considerations for the public: By using one of our public licenses, a licensor grants the public permission to use the licensed material under specified terms and conditions. If the licensor’s permission is not necessary for any reason–for example, because of any applicable exception or limitation to copyright–then that use is not regulated by the license. Our licenses grant only permissions under copyright and certain other rights that a licensor has authority to grant. Use of the licensed material may still be restricted for other reasons, including because others have copyright or other rights in the material. A licensor may make special requests, such as asking that all changes be marked or described. Although not required by our licenses, you are encouraged to respect those requests where reasonable.

Creative Commons Attribution-ShareAlike 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution-ShareAlike 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.
b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.
c. **BY-SA Compatible License** means a license listed at creativecommons.org/compatiblelicenses, approved by Creative Commons as essentially the equivalent of this Public License.
d. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.
e. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.
f. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.
g. **License Elements** means the license attributes listed in the name of a Creative Commons Public License. The License Elements of this Public License are Attribution and ShareAlike.
h. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.
i. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.
j. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.
k. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in

ways that members of the public may access the material from a place and at a time individually chosen by them.

l. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

m. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. **License grant**.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:
   A. reproduce and Share the Licensed Material, in whole or in part; and
   B. produce, reproduce, and Share Adapted Material.
2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.
3. Term. The term of this Public License is specified in Section 6(a).
4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.
5. Downstream recipients.
   A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.
   B. Additional offer from the Licensor – Adapted Material. Every recipient of Adapted Material from You automatically receives an offer from the Licensor to exercise the Licensed Rights in the Adapted Material under the conditions of the Adapter's License You apply.
   C. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or

sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. **Other rights**.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
2. Patent and trademark rights are not licensed under this Public License.
3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. **Attribution**.

1. If You Share the Licensed Material (including in modified form), You must:

   A. retain the following if it is supplied by the Licensor with the Licensed Material:
      i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);
      ii. a copyright notice;
      iii. a notice that refers to this Public License;
      iv. a notice that refers to the disclaimer of warranties;
      v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;
   B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and
   C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.
2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.
3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

b. **ShareAlike**.
In addition to the conditions in Section 3(a), if You Share Adapted Material You produce, the following conditions also apply.

1. The Adapter's License You apply must be a Creative Commons license with the same License Elements, this version or later, or a BY-SA Compatible License.
2. You must include the text of, or the URI or hyperlink to, the Adapter's License You apply. You may satisfy this condition in any reasonable manner based on the medium, means, and context in which You Share Adapted Material.
3. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, Adapted Material that restrict exercise of the rights granted under the Adapter's License You apply.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;
b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material, including for purposes of Section 3(b); and
c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. **Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.**
b. **To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of**

this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.
b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

   1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or
   2. upon express reinstatement by the Licensor.

   For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.
c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.
d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.
b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.
b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.
d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the “Licensor.” The text of the Creative Commons public licenses is dedicated to the public domain under the CC0 Public Domain Dedication. Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark “Creative Commons” or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.

Additional languages available: Bahasa Indonesia, Español, euskara, Deutsch, Español, français, hrvatski, italiano, latviski, Lietuvių, Nederlands, norsk, polski, português, suomeksi, svenska, te reo Māori, Türkçe, čeština, Ελληνικά, русский, українська, العربية, 日本語, 한국어. Please read the FAQ for more information about official translations.