1. 语言概述

Delphi 是一种高级、编译、强类型语言，支持结构化和面向对象编程。基于 Delphi，其优点包括易于阅读的代码、快速编译以及使用多个单元文件进行模块化编程。Delphi 具有支持 RAD Studio 组件框架和环境的特殊功能。在大多数情况下，本语言指南中的描述和示例假定您正在使用 Embarcadero 开发工具。

大多数使用 Embarcadero 软件开发工具的开发人员在集成开发环境（IDE）中编写和编译代码。Embarcadero 开发工具处理设置项目和源文件的许多细节，例如维护单元之间的依赖信息。该产品还对程序组织施加了严格来说不是 Delphi 语言规范一部分的约束。例如，Embarcadero 开发工具强制实施某些文件和程序命名约定，如果在 IDE 外部编写程序并从命令提示符编译它们，则可以避免这些约定。

本语言指南通常假定您正在使用 IDE，并且正在构建使用可视组件库 （VCL） 的应用程序。然而，有时 Delphi 特定的规则与适用于所有 Delphi 编程的规则是有区别的。

本节涵盖以下主题：

程序组织：介绍允许您将应用程序划分为单元和命名空间的基本语言功能。

示例程序：显示了控制台和 GUI 应用程序的小示例，以及有关从命令行运行编译器的基本说明。

程序组织

Delphi 程序通常分为称为单元的源代码模块。大多数程序都以程序标题开头，该标题指定程序的名称。程序标题后跟一个可选的 use 子句，然后是一个声明和语句块。use 子句列出了链接到程序的单元;这些单元可以由不同的程序共享，通常有自己的 use 子句。

uses 子句为编译器提供有关模块之间依赖关系的信息。由于此信息存储在模块本身中，因此大多数 Delphi 语言程序不需要 makefile、头文件或预处理器“include”指令。

delphi 源码文件

编译器希望在三种文件中找到 Delphi 源代码：

单元源文件（以 .pas 扩展名结尾）

项目文件（以 .dpr 扩展名结尾）

包源文件（以 .dpk 扩展名结尾）

单元源文件通常包含应用程序中的大部分代码。每个应用程序都有一个项目文件和多个单元文件;项目文件对应于传统 Pascal 中的程序文件，将单元文件组织到应用程序中。Embarcadero 开发工具自动为每个应用程序维护一个项目文件。

如果要从命令行编译程序，则可以将所有源代码放入单元 （.pas） 文件中。如果使用 IDE 生成应用程序，它将生成一个项目 （.dpr） 文件。

包源文件类似于项目文件，但它们用于构造称为包的特殊动态可链接库。

其它文件

除了源代码模块，Embarcadero 产品还使用几个非 Pascal 文件来构建应用程序。这些文件由 IDE 自动维护，包括

VCL 窗体文件（在 Win32 上具有 .dfm 扩展名）

资源文件（以 .res 结尾）

项目选项文件（以 .dof 结尾）

VCL 窗体文件包含窗体的属性及其拥有的组件的说明。每个窗体文件表示单个窗体，该窗体通常对应于应用程序中的窗口或对话框。IDE 允许您以文本形式查看和编辑样式文件，并将样式文件另存为文本（一种非常适合版本控制的格式）或二进制文件。尽管默认行为是将表单文件另存为文本，但通常不会手动编辑它们;为此目的，使用 Embarcadero 的视觉设计工具更为常见。每个项目至少有一个表单，每个表单都有一个关联的单元 （.pas） 文件，默认情况下，该文件与表单文件具有相同的名称。

除了 VCL 窗体文件之外，每个项目还使用资源 （.res） 文件来保存应用程序的图标和其他资源（如字符串）。默认情况下，此文件与项目 （.dpr） 文件同名。

项目选项 （.dof） 文件包含编译器和链接器设置、搜索路径信息、版本信息等。每个项目都有一个与项目 （.dpr） 文件同名的关联项目选项文件。通常，此文件中的选项是从“项目选项”对话框中设置的。

IDE 中的各种工具将数据存储在其他类型的文件中。桌面设置 （.dsk） 文件包含有关窗口排列和其他配置选项的信息;桌面设置可以是特定于项目的，也可以是环境范围的。这些文件对编译没有直接影响。

编译器生成的文件

首次生成应用程序或包时，编译器会为项目中使用的每个新单元生成一个已编译的单元文件（Win32 上的 .dcu）;然后链接项目中的所有 .dcu 文件以创建单个可执行文件或共享包。首次生成包时，编译器会为包中包含的每个新单元生成一个文件，然后创建 .dcp 和包文件。如果使用 GD 编译器开关，链接器将生成映射文件和 .drc 文件;包含字符串资源的 .drc 文件可以编译为资源文件。

生成项目时，除非自上次编译以来其源 （.pas） 文件已更改，找不到其 .dcu/.dpu 文件，显式告诉编译器重新处理它们，或者单元的接口依赖于已更改的另一个单元，否则不会重新编译单个单元。事实上，单元的源文件根本不需要存在，只要编译器可以找到已编译的单元文件，并且该单元不依赖于其他已更改的单元。

示例程序

下面的示例说明了 Delphi 编程的基本功能。这些示例显示了通常不会从 IDE 编译的简单应用程序;您可以从命令行编译它们。

控制台程序

下面的程序是一个简单的控制台应用程序，您可以从命令提示符编译和运行：

```
program Greeting;

 {$APPTYPE CONSOLE}

 var
    MyMessage: string;

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

第一行声明一个名为 Greeting 的程序。{$APPTYPE CONSOLE} 指令告诉编译器这是一个控制台应用程序，要从命令行运行。下一行声明一个名为 MyMessage 的变量，该变量包含字符串。（Delphi 具有真正的字符串数据类型。然后，程序将字符串“Hello world！”分配给变量 MyMessage，并使用 Writeln 过程将 MyMessage 的内容发送到标准输出。（Writeln 在系统单元中隐式定义，编译器会自动将其包含在每个应用程序中。

您可以将此程序键入名为 greeting.pas 或 greeting.dpr 的文件中，并通过输入以下内容对其进行编译：

```
DCC32 greeting
```

以生成 Win32 可执行文件。

运行生成的可执行文件打印消息 你好世界！

除了简单之外，此示例在几个重要方面与您可能使用 Embarcadero 开发工具编写的程序不同。首先，它是一个控制台应用程序。Embarcadero 开发工具最常用于编写具有图形界面的应用程序;因此，您通常不会称 Writeln。此外，整个示例程序（除了 Writeln）都在一个文件中。在典型的 GUI 应用程序中，示例第一行标题的程序将放置在单独的项目文件中，该文件不包含任何实际的应用程序逻辑，除了对单元文件中定义的例程的几次调用。

更复杂的控制台程序

下一个示例显示一个分为两个文件的程序：项目文件和单元文件。可以另存为 greeting.dpr 的项目文件如下所示：

```
 program Greeting;

 {$APPTYPE CONSOLE}
```

```
uses
  Unit1;

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

第一行声明一个名为 greeting 的程序，该程序再次是一个控制台应用程序。uses Unit1;子句告诉编译器程序问候语依赖于名为 Unit1 的单元。最后，程序调用 PrintMessage 过程，将字符串 Hello World！打印消息过程在 Unit1 中定义。以下是 Unit1 的源代码，必须保存在名为 Unit1.pas 的文件中：

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation

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

end.
```

Unit1 定义了一个名为 PrintMessage 的过程，该过程将单个字符串作为参数，并将字符串发送到标准输出。（在 Delphi 中，不返回值的例程称为过程。返回值的例程称为函数。

请注意，PrintMessage 在 Unit1 中声明了两次。保留字接口下的第一个声明使 PrintMessage 可用于使用 Unit1 的其他模块（如问候语）。第二个声明，在保留字实现下，实际上定义了 PrintMessage。

您现在可以从命令行编译问候语，方法是输入

```
DCC32 greeting
```

以生成 Win32 可执行文件。

无需包含 Unit1 作为命令行参数。当编译器处理 greeting.dpr 时，它会自动查找问候程序所依赖的单元文件。生成的可执行文件执行与我们的第一个示例相同的操作：它打印消息 Hello world！

VCL 程序

下一个示例是使用 IDE 中的可视组件库 （VCL） 组件生成的应用程序。此程序使用自动

生成的表单和资源文件，因此您无法仅从源代码编译它。但它说明了 Delphi 语言的重要特征。除了多个单元之外，该程序还使用类和对象。

该程序包括一个项目文件和两个新的单元文件。一、项目文件：

```
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.
```

再一次，我们的程序被称为 Greeting。它使用三个单元：Forms，它是 VCL 的一部分;Unit1，与应用程序的主窗体 （Form1） 相关联;Unit2，它与另一个表单 （Form2） 相关联。

该程序对名为 Application 的对象进行一系列调用，该对象是在 Forms 单元中定义的 Vcl.Forms.TApplication 类的实例。（每个项目都有一个自动生成的应用程序对象。其中两个调用调用名为 CreateForm 的 Vcl.Forms.TApplication 方法。对 CreateForm 的第一次调用将创建 Form1，它是在 Unit1 中定义的 TForm1 类的实例。对 CreateForm 的第二次调用创建 Form2，这是在 Unit2 中定义的 TForm2 类的实例。

Unit1 看起来像这样：

```
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 创建一个名为 TForm1 的类（派生自 Vcl.Forms.TForm）和此类 Form1 的实例。TForm1 类包括一个按钮 - Button1，Vcl.StdCtrls.TButton 的一个实例 - 以及一个名为 Button1Click 的过程，当用户按下 Button1 时调用。Button1Click 隐藏 Form1 并显示 Form2（对 Form2.ShowModal 的调用）。

注意：在前面的示例中，Form2.ShowModal 依赖于使用自动创建的表单。虽然这对于示例代码很好，但强烈建议不要使用自动创建的表单。

Form2 在 Unit2 中定义：

```
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 创建一个名为 TForm2 的类和该类的一个实例 Form2。TForm2 类包括一个按钮（CancelButton，Vcl.StdCtrls.TButton 的实例）和一个标签（Label1，Vcl.StdCtrls.TLabel 的实例）。您无法从源代码中看到这一点，但 Label1 显示一个标题，上面写着 Hello world！标题在 Form2 的表单文件 Unit2.dfm 中定义。

TForm2 声明并定义了一个方法 CancelButtonClick，每当用户按下 CancelButton 时，就会在运行时调用该方法。此过程（以及 Unit1 的 TForm1.Button1Click）称为事件处理程序，因为它响应程序运行时发生的事件。事件处理程序由 Form1 和 Form2 的窗体文件分配给特定事件。

当 Greeting 程序启动时，将显示 Form1，而 Form2 不可见。（默认情况下，在运行时仅对项目文件中创建的第一个窗体可见。这称为项目的主窗体。当用户按下 Form1 上的按钮时，Form2 将显示 Hello world！问候。当用户按标题栏上的“取消按钮”或“关闭”按钮时，Form2 将关闭。

2. 程序和单元

本主题介绍 Delphi 应用程序的整体结构：程序头、单元声明语法和 uses 语句。

- ✓ 将大型程序划分为可单独编辑的模块。
- ✓ 创建可在程序之间共享的库。
- ✓ 将库分发给其他开发人员，而无需提供源代码。

程序和单元

程序结构和语法

一个完整的、可执行的 Delphi 应用程序由多个单元模块组成，所有这些模块都由称为项目文件的单个源代码模块捆绑在一起。在传统的 Pascal 编程中，所有源代码，包括主程序，都存储在.pas 文件中。Embarcadero 工具使用文件扩展名.dpr 来指定主程序源模块，而大多数其他源代码驻留在具有传统.pas 扩展名的单元文件中。若要生成项目，编译器需要项目源文件，以及每个单元的源文件或已编译单元文件。

注意：严格来说，您不需要在项目中使用任何单元，但所有程序都会自动使用 System 单元和 SysInit 单元。

可执行 Delphi 应用程序的源代码文件包含：

- ✓ 程序标题头，
- ✓ 一个 use 语句（可选），以及

- ✓ 声明和可执行语句块。

编译器（因此是 IDE）希望在单个项目 （.dpr） 文件中找到这三个元素。

程序标题头

程序标题指定可执行程序的名称。它由保留字 program 组成，后跟有效标识符，后跟分号。对于使用 Embarcadero 工具开发的应用程序，标识符必须与项目源文件名匹配。

下面的示例演示名为 Editor 的程序的项目源文件。由于该程序称为 Editor，因此此项目文件称为 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.
```

第一行包含程序标题。此示例中的 uses 子句指定对三个附加单元的依赖关系：Forms、REAbout 和 REMain。$R 编译器指令将项目的资源文件链接到程序中。最后，开始和结束关键字之间的语句块在程序运行时执行。与所有 Delphi 源文件一样，项目文件以句点（不是分号）结尾。

Delphi 项目文件通常很短，因为程序的大部分逻辑都驻留在其单元文件中。Delphi 项目文件通常只包含足够的代码来启动应用程序的主窗口，并启动事件处理循环。项目文件由 IDE 自动生成和维护，很少需要手动编辑它们。

在标准 Pascal 中，程序标题可以在程序名称后包含参数：

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

Embarcadero 的 Delphi 忽略了这些参数。

在 RAD Studio 中，程序标题引入了自己的命名空间，称为项目默认命名空间。

Uses 语句

uses 子句列出了合并到程序中的那些单元。这些单元可能又有自己的 uses 语句。有关单元源文件中的 use 子句的详细信息，请参阅下面的《单元引用和 uses 子句》。

uses 子句由关键字 uses 组成，后跟项目文件直接依赖的逗号分隔单元列表。

语句块

该块包含程序运行时执行的简单或结构化语句。在大多数程序文件中，块由一个复合语句组成，该复合语句括在保留字 begin 和 end 之间，其组件语句只是对项目的 Application 对象的方法调用。大多数项目都有一个全局 Application 变量，其中包含 Vcl.Forms.TApplication、Web.WebBroker.TWebApplication 或 Vcl.SvcMgr.TServiceApplication 的实例。该块还可以包含常量、类型、变量、过程和函数的声明;这些声明必须位于块的语句部分之前。请注意，表示程序源结束的结尾必须后跟句点 （.）：

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

单元结构和语法

单元由类型（包括类）、常量、变量和例程（函数和过程）组成。每个单元都在其自己的源 （.pas） 文件中定义。

单元文件以单元标题开头，后跟接口关键字。在接口关键字之后，use 子句指定单元依赖项的列表。接下来是实现部分，然后是可选的初始化和销毁部分。骨架单元源文件如下所示：

```
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.
```

该单元必须以保留字结尾，后跟句点结束。

单元头

单元标题指定单元的名称。它由保留字 unit 组成，后跟有效标识符，后跟分号。对于使用 Embarcadero 工具开发的应用程序，标识符必须与单元文件名匹配。因此，单元标题：

```
unit MainForm;
```

将出现在名为 MainForm.pas 的源文件中，包含编译单元的文件将是 MainForm.dcu。单元名称在项目中必须是唯一的。即使它们的单元文件位于不同的目录中，两个同名的单元也不能在单个程序中使用。

接口部分

单元的接口部分从保留字 interface 开始，一直持续到实现部分的开头。接口部分声明可供客户端使用的常量、类型、变量、过程和函数。也就是说，对于希望使用该单元中的元素的其他单元或程序。这些实体称为公共实体，因为其他单元中的代码可以访问它们，就像它们是在单元本身中声明的一样。

过程或函数的接口声明仅包含例程的签名。即例程的名称、参数和返回类型（对于函数）。包含过程或函数的可执行代码的块位于实现部分。因此，接口部分中的过程和函数声明的工作方式类似于前向声明。

类的接口声明必须包括所有类成员的声明：字段、属性、过程和函数。

接口部分可以包含其自己的 uses 子句，该子句必须紧跟在关键字 interface 之后。

实现部分

单元的实现部分从保留字 implementation 开始，一直持续到初始化部分的开头，如果没有初始化部分，则一直持续到单元的末尾。实现部分定义在接口部分中声明的过程和函数。在实现部分中，可以按任意顺序定义和调用这些过程和函数。在实现部分中定义参数列表时，可以从公共过程和函数标题中省略参数列表;但是，如果包含参数列表，则它必须与接口部分中的声明完全匹配。

除了公共过程和函数的定义之外，实现部分还可以声明单元专用的常量、类型（包括类）、变量、过程和函数。也就是说，与接口部分不同，其他单元无法访问在实现部分中声明的实体。

实现部分可以包含其自己的 uses 子句，该子句必须紧跟在关键字 implementation 之后出现。在实现部分中指定的单元中声明的标识符只能在实现部分本身中使用。您不能在接口部分中引用此类标识符。

初始化部分

初始化部分是可选的。它从保留字 initialization 开始，一直持续到销毁部分的开始，如果没有销毁部分，则一直持续到单元结束。初始化部分包含在程序启动时按出现的顺序执行的语句。因此，例如，如果您定义了需要初始化的数据结构，则可以在初始化部分中执行此操作。

对于接口 uses 列表中的单元，客户端使用的单元的初始化部分将按照单元在客户端的 uses 子句中出现的顺序执行。

较老的“begin...end“语法仍然有效。基本上，保留字“begin”可以用来代替初始化，后跟零个或多个执行语句。使用较旧的“begin...end“语法无法指定销毁部分。在这种情况下，通过向 ExitProc 变量提供过程来完成。不建议将此方法用于以后的代码，但您可能会看到它在较旧的源代码中使用。

销毁部分

销毁部分是可选的，只能出现在具有初始化部分的单元中。终结部分从保留字 finalization 开始，一直持续到单元结束。它包含在主程序终止时执行的语句（除非使用 Halt 过程终止程序）。使用销毁部分释放在初始化部分中分配的资源。

销毁部分的执行顺序与初始化部分的顺序相反。例如，如果应用程序按该顺序初始化单元 A、B 和 C，它将按顺序 C、B 和 A 完成它们。

一旦单元的初始化代码开始执行，相应的销毁部分保证在应用程序关闭时执行。因此，销毁部分必须能够处理未完全初始化的数据，因为如果发生运行时错误，初始化代码可能无法完全执行。

单元引用和 uses 子句

uses 子句列出出现子句的程序、库或单元所使用的单元。uses 子句可以出现在

- ✓ 程序或库的项目文件
- ✓ 单元 的接口部分
- ✓ 单元的执行部分

大多数项目文件都包含 uses 子句，大多数单元的接口部分也是如此。单元的实现部分也可以包含其自己的 uses 子句。

System 单元和 SysInit 单元由每个应用程序自动使用，不能在 uses 子句中显式列出。（系统实现文件 I/O、字符串处理、浮点运算、动态内存分配等的例程。其他标准库单元（如 SysUtils）必须显式包含在 uses 子句中。在大多数情况下，在项目中添加和删除单元时，IDE 会将所有必需的单元放在 uses 子句中。

区分大小写：在单元声明和 uses 子句中，单元名称必须与文件名匹配（大小写）。在其他上下文（如限定标识符）中，单元名称不区分大小写。为避免单元引用出现问题，请明确引用单元源文件：

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

如果项目文件中出现这样的显式引用，则其他源文件可以使用不需要匹配大小写的简单

uses 子句引用该单元：

uses Myunit;

Uses 语法

uses 子句由保留字 uses 组成，后跟一个或多个逗号分隔的单元名称，后跟分号。例子：
uses Forms, Main;

```
uses
  Forms,
  Main;
```

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

在程序或库的 uses 子句中，任何单元名称都可以后跟保留字 in 和源文件的名称，带或不带目录路径，用单引号引起来;目录路径可以是绝对路径，也可以是相对路径。例子：

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

当您需要指定单元的源文件时，请在单元名称后使用关键字 in。由于 IDE 希望单元名称与其所在的源文件的名称匹配，因此通常没有理由这样做。仅当源文件的位置不清楚时才需要使用 in，例如：

- ✓ 您使用的源文件与项目文件位于不同的目录中，并且该目录不在编译器的搜索路径中。
- ✓ 编译器搜索路径中的不同目录具有相同名称的单元。
- ✓ 您正在从命令行编译控制台应用程序，并且您使用与其源文件的名称不匹配的标识符命名了单元。

编译器还依赖于 in 以确定哪些单元是项目的一部分。只有出现在项目（.dpr）文件的 uses 子句中，后跟 in 和文件名的单元才被视为项目的一部分;uses 中的其他单元由项目使用，但不属于该项目。这种区别对编译没有影响，但它会影响 IDE 工具，如项目管理器。

在单元的 uses 子句中，不能使用 in 告诉编译器在哪里可以找到源文件。每个单元都必须位于编译器的搜索路径中。此外，单元名称必须与其源文件的名称匹配。

多个单元和间接单元引用

单元在 uses 子句中的出现顺序决定了其初始化的顺序，并影响编译器定位标识符的方式。如果两个单元声明具有相同名称的变量、常量、类型、过程或函数，则编译器将使用 uses 子句中最后列出的单元中的单元。（要从其他单元访问标识符，您必须添加一个限定符：UnitName.Identifier。

uses 子句只需要包括出现子句的程序或单元直接使用的单元。也就是说，如果单元 A 引用在单元 B 中声明的常量、类型、变量、过程或函数，则 A 必须显式使用 B。如果 B 反过

来引用单元 C 的标识符，则 A 间接依赖于 C;在这种情况下，C 不需要包含在 A 中的 use 子句中，但编译器仍然必须能够找到 B 和 C 才能处理 A。

以下示例说明了间接依赖关系：

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

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

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

在此示例中，Prog 直接依赖于 Unit2，而 Unit2 直接依赖于 Unit1。因此，Prog 间接依赖于 Unit1。由于 Unit1 未出现在 Prog 的使用子句中，因此 Unit1 中声明的标识符对 Prog 不可用。

若要编译客户端模块，编译器需要直接或间接找到客户端依赖的所有单元。但是，除非这些单元的源代码已更改，否则编译器只需要其 .dcu 文件，而不需要其源 （.pas） 文件。

在单元的接口部分进行更改时，必须重新编译依赖于该更改的其他单元。但是，当仅在单元的实现或其他部分中进行更改时，不必重新编译依赖单元。编译器会自动跟踪这些依赖项，并仅在必要时重新编译单元。

单元循环引用

当单元直接或间接相互引用时，这些单元被称为相互依赖的。只要没有循环路径将一个接口部分的 uses 子句连接到另一个接口部分的 uses 子句，就允许相互依赖。换句话说，从单元的接口部分开始，永远不可能通过其他单元的接口部分遵循引用返回到该单元。要使相互依赖的模式有效，每个循环引用路径必须通过至少一个实现节的 uses 子句。

在两个相互依赖的单元的最简单情况下，这意味着这些单元不能在其接口 uses 子句中相互列出。因此，以下示例会导致编译错误：

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

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

但是，如果将其中一个引用移动到实现部分，则这两个单元可以合法地相互引用：

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

unit Unit2;
interface
//...

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

为了减少循环引用的机会，最好尽可能在实现 uses 子句中列出单元。只有在接口部分中使用来自另一个单元的标识符时，才有必要在接口 uses 子句中列出该单元。

命名空间

注意： RAD Studio 现在除了支持命名空间或单元名称外，还支持单元范围名称或前缀。为了使名称被视为完全限定，必须包括单元范围名称。有关更多详细信息，请参阅《单元范围名称》。

在 Delphi 中，单元是类型的基本容器。在 Delphi 中，命名空间是 Delphi 单元的容器。

与传统的 Delphi 单元不同，命名空间可以嵌套以形成包含层次结构。嵌套命名空间提供了一种组织标识符和类型的方法，并用于消除具有相同名称的类型。由于它们是 Delphi 单元的容器，因此命名空间也可用于区分驻留在不同包中的同名单元。

例如，MyNameSpace 中的类 MyClass 与 YourNamespace 中的类 MyClass 不同。

本主题介绍以下内容：

- ✓ 项目默认命名空间和命名空间声明。
- ✓ 命名空间搜索范围。
- ✓ 在 Delphi 单元中使用命名空间。

声明命名空间

在 RAD Studio 中，项目文件（程序、库或包）隐式引入自己的命名空间，称为项目默认命名空间。单元可以是项目默认命名空间的成员，也可以显式声明自己是不同命名空间的成员。在任一情况下，单元都会在其单元标头中声明其命名空间成员身份。例如，请考虑以下显式命

名空间声明：

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

首先，请注意命名空间的各个部分由点分隔。命名空间不会为点之间的标识符引入新符号;圆点是单元名称的一部分。此示例的源文件名为 MyCompany.MyWidgets.MyUnit.pas，编译的输出文件名为 MyCompany.MyWidgets.MyUnit.dcu。

其次，请注意，这些点表示一个命名空间在另一个命名空间中的概念嵌套或包含。上面的示例将单元 MyUnit 声明为 MyWidgets 命名空间的成员，该命名空间本身包含在 MyCompany 命名空间中。同样，应该注意的是，此包含仅用于文档目的。

项目默认命名空间为项目中的所有单元声明一个命名空间。请考虑以下声明：

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

这些语句分别为程序、库和包建立项目默认命名空间。命名空间是通过从声明中删除最右侧的标识符（和点）来确定的。

省略显式命名空间的单元称为泛型单元。通用单元自动成为项目默认命名空间的成员。给定前面的程序声明，下面的单元声明将导致编译器将 MyUnit 视为 MyCompany.Programs 命名空间的成员。

```
unit MyUnit;
```

项目默认命名空间不会影响通用单元的 Delphi 源文件的名称。在前面的示例中，Delphi 源文件名为 MyUnit.pas。相同的规则适用于 dcu 文件名。当前示例中生成的 dcu 文件将是 MyUnit.dcu。

命名空间字符串不区分大小写。编译器认为两个命名空间仅在大小写相同时才不同。但是，编译器确实保留了命名空间的大小写，并将在输出文件名、错误消息和 RTTI 单元标识符中使用保留的大小写。类和类型名称的 RTTI 将包括完整的命名空间规范。

搜索命名空间

一个单元必须声明它所依赖的其他单元。编译器必须在这些单元中搜索标识符。对于显式命名空间中的单元，搜索范围是已知的，但对于泛型单元，编译器必须建立命名空间搜索范围。

请考虑以下单元和 uses 语句：

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

这些声明将 MyUnit1 建立为 MyCompany.ProjectX.ProgramY 命名空间的成员。MyUnit1 依赖于其他三个单元：MyCompany.Libs.Unit2 和通用单元 Unit3 和 Unit4。编译器可以在 Unit2 中解析标识符名称，因为 uses 子句指定了完全限定的单元名称。若要解析 Unit3 和 Unit4 中的标识符名称，编译器必须建立命名空间搜索顺序。

搜索顺序

搜索位置可以来自三个可能的源：编译器选项、项目默认命名空间和当前单元的命名空间。编译器按以下顺序解析标识符名称：

- ✓ 当前单元命名空间（如果有）
- ✓ 项目默认命名空间（如果有）
- ✓ 编译器选项指定的命名空间。

搜索例子

以下示例项目和单元文件演示了命名空间搜索顺序：

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

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

给定此程序示例，编译器将按以下顺序搜索命名空间：

1. MyCompany.ProjectX.ProgramY
2. MyCompany.ProjectX
3. 编译器选项指定的命名空间。

请注意，如果当前单元是泛型的（即，它的 unit 语句中没有显式命名空间声明），则解析从项目默认命名空间开始。

使用命名空间

Delphi 的 uses 子句将模块带入当前单元的上下文中。uses 子句必须通过模块的完全限定名称（即包括完整的命名空间规范）或泛型名称来引用模块，从而依赖于命名空间解析机制来定位单元。

全限定单元名称

下面的示例演示了带有命名空间的 uses 子句：

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

将模块引入上下文后，源代码可以通过非限定名称或完全限定名称引用该模块中的标识符（如有必要，以消除不同单元中具有相同名称的标识符的歧义）。以下 Writeln 语句是等效的：

```
uses MyCompany.Libs.Unit2;

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

完全限定标识符必须包含完整的命名空间规范。在前面的示例中，仅使用命名空间的一部分引用 SomeString 将是一个错误：

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

在 uses 子句中仅引用命名空间的一部分也是一个错误。没有导入命名空间中所有单位和符号的机制。以下代码不会导入 MyCompany 命名空间中的所有单位和符号：

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

此限制也适用于 with-do 语句。以下内容将产生编译器错误：

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

多单元命名空间

如果单元声明引用同一命名空间，则多个单元可以属于同一命名空间。例如，可以使用以下单元声明创建两个文件，unit1.pas 和 unit2.pas：

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

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

在此示例中，命名空间 MyCompany.ProjectX.ProgramY 在逻辑上包含来自 unit1.pas 和 unit2.pas 的所有接口符号。

命名空间中的符号名称在命名空间中的所有单元中必须是唯一的。在上面的示例中，Unit1 和 Unit2 都定义了一个名为 mySymbol 的全局接口符号是一个错误。

命名空间中聚合的各个单元不可用于源代码，除非在文件的 uses 子句中显式使用了各个单元。换句话说，如果源文件仅使用命名空间，则引用该命名空间中单元中的符号的完全限定标识符表达式必须使用命名空间名称，而不仅仅是定义该符号的单元的名称。

uses 子句可以引用命名空间以及该命名空间中的单个单元。在这种情况下，引用 uses 子句中列出的特定单元的符号的完全限定表达式可以使用限定符的实际单元名称或限定符的完全限定名称（包括命名空间和单元名称）来引用。这两种形式的参照是相同的，指的是同一个符号。

注： 仅在从源文件或 dcu 文件编译时，在 uses 子句中显式使用单元才有效。如果将命名空间单元编译为程序集，并且程序集由项目引用而不是单个单元，则显式引用命名空间中单元的源代码将失败。

单元范围名称

单元范围名称是附加到 RAD Studio 库 （VCL-FMX-RTL）单元名称前面的前缀。也就是说，单元、函数、类和成员的名称在单元名称前面有一个单元作用域名称，如下所示：

<unitscope>.<unitname>. ...

System.SysUtils

FMX.Controls

- ✓ 将单元分类为基本组，例如 Vcl、System、FMX 等。
- ✓ 确保使用 IDE 编写的代码的兼容性。
- ✓ 区分名称不明确的成员（即，当成员的名称与另一个单元的成员名称匹配时，确保正确的名称解析）。
- ✓ 通常以单个大写字母开头，后跟小写字母（如 Data）。
- ✓ 通常由一个元素（如 DataSnap）组成，尽管有些元素由两个元素组成（如 System.Generics）。

完全限定名称必须包括单元范围名称：

System.Classes.TStream

3. 基础语法元素

基础语法元素

基本语法元素（称为标记）组合在一起以形成表达式、声明和语句。语句描述可以在程序中执行的算法操作。表达式是出现在语句中并表示值的语法单元。声明定义可在表达式和语句中使用的标识符（如函数或变量的名称），并在适当的情况下为标识符分配内存。

本主题介绍 Delphi 语言字符集，并描述用于声明以下内容的语法：

- ✓ 标识符
- ✓ 数字

- ✓ 字符串
- ✓ 标签
- ✓ 源代码注释

字符范围

Delphi 语言对其字符集使用 Unicode 字符编码，包括字母和字母数字 Unicode 字符以及下划线。Delphi 不区分大小写。空格字符和控制字符（U+0000 到 U+001F，包括 U+000D，回车符或行尾字符）为空白。

如果文件包含字节顺序标记，RAD Studio 编译器将接受以 UCS-2 或 UCS-4 编码的文件。但是，编译速度可能会因使用 UTF-8 以外的格式而受到影响。UCS-4 编码源文件中的所有字符必须在 UCS-2 中表示，没有代理项对。仅当指定了代码页编译器选项时，才接受具有代理项对（包括 GB18030）的 UCS-2 编码。

标记

在最简单的层面上，程序是由分隔符分隔的标记序列。标记是程序中最小的有意义的文本单元。分隔符可以是空白或注释。严格来说，并不总是需要在两个标记之间放置分隔符;例如，代码片段：

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

是完全合法的。然而，约定和可读性决定了我们用两行写，如：

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

标记分为特殊符号、标识符、保留字、指令、数字、标签和字符串。仅当标记是字符串时，分隔符才能成为标记的一部分。相邻标识符、保留字、数字和标签之间必须有一个或多个分隔符。

特殊符号

特殊符号是具有固定含义的非字母数字字符或此类字符对。以下单个字符是特殊符号：

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

以下字符对也是特殊符号：

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

下表显示了 Delphi 中使用的具有相似含义的符号对（符号对 {} 和 （* *） 是注释字符，在注释和编译器指令中进一步描述）：

特殊符号	相似特殊符号
[ ]	(. .)
{ }	(* *)

左括号 [类似于左括号和句点 (.

右括号] 类似于句点和右括号的字符对 .)

左大括号 { 类似于左括号和星号 (*

右大括号 } 类似于星号和右括号的字符对*)

注意：%， ?， \， !， “ （双引号）， _（下划线）， |（管道）和 ~（波浪号）不是特殊符号。

标识符

标识符表示常量、变量、字段、类型、属性、过程、函数、程序、单元、库和包。标识符可以是任意长度，但只有前 255 个字符是重要的。标识符必须以字母字符、Unicode 字符或下划线 （_） 开头，并且不能包含空格。第一个字符后允许使用字母数字字符、Unicode 字符、数字和下划线。保留字不能用作标识符。由于 Delphi 语言不区分大小写，因此像 CalculateValue 这样的标识符可以用以下任何一种方式编写：

CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE

由于单元名称与文件名相对应，因此大小写中的不一致有时会影响编译。有关详细信息，请参阅程序和单元 （Delphi） 中的单元引用和使用子句部分。

限定标识符

使用已在多个位置声明的标识符时，有时需要限定标识符。限定标识符的语法为：

identifier1.identifier2

其中 identifier1 限定 identifier2。例如，如果两个单元分别声明一个名为 CurrentValue 的变量，则可以通过编写以下内容来指定要访问 Unit2 中的 CurrentValue：

```
Unit2.CurrentValue
```

可以迭代限定符。例如：

```
Form1.Button1.Click
```

调用 Form1 的 Button1 中的 Click 方法。

如果您不限定标识符，则其解释由声明和语句 （Delphi） 中的块和范围中所述的范围规则确定。

扩展标识符

您可能会遇到与 Delphi 语言保留字同名的标识符（例如类中的类型或方法）。例如，一个类可能有一个名为 begin 的方法。Delphi 保留字（如 begin）不能用于标识符名称。

如果完全限定标识符，则没有问题。例如，如果要对标识符名称使用 Delphi 保留字类型，则必须使用其完全限定名称：

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

作为较短的替代方法，与号 （&） 运算符可用于解决标识符和 Delphi 语言保留字之间的歧义。&阻止将关键字解析为关键字（即保留字）。如果遇到与 Delphi 关键字同名的方法或类型，则可以省略命名空间规范，前提是在标识符名称前面加上与号。但是，当您声明与关键字同名的标识符时，必须使用 &：

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

保留字

以下保留字不能重新定义或用作标识符。

and	end	interface	record	var
array	except	is	repeat	while
as	exports	label	resourcestring	with
asm	file	library3	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	

注意：除了上表中的保留字之外，private, protected, public, published, and automated 在类类型声明中充当保留字，但在其他情况下被视为指令。at 和 on 也有特殊含义，应视为保留字。object 用于定义方法指针。

指令字

Delphi 有不止一种类型的指令。“指令”的一个含义是在源代码中的特定位置敏感的词。这种类型的指令在 Delphi 语言中具有特殊含义，但与保留字不同，它只出现在用户定义的标识符无法出现的上下文中。因此，尽管不建议这样做，但您可以定义一个看起来与指令完全相同的标识符。

absolute	export[12]	name	public	stdcall
abstract	external	near[1]	published	strict
assembler[12]	far1	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[311]	platform[11]	safecall	
dynamic	local[4]	private	sealed[5]	
experimental[11]	message	protected	static	

注意：

1. far, near, and resident 已经过时。
2. inline 在过程和函数声明的末尾使用指令来标记内联的过程或函数，但成为 Turbo Pascal 的保留词。
3. 当用作项目源代码中的第一个标记时，library 也是一个关键字;它指示 DLL 目标。否则，它会标记一个符号，以便在使用时生成库警告。
4. local 是 Kylix 指令，对于 Win32 的 Delphi 来说被忽略了。

5. sealed 是一个语法奇怪的类指令：“类密封”。密封类不能扩展或派生（如 C++ 中的 final）。
6. winapi 定义默认平台调用约定。例如，在 Win32 上，winapi 与 stdcall 相同。
7. package 用作第一个标记时，指示包目标并启用包语法。require 和 contains 指令仅在包语法中。
8. helper 指示“类帮助程序”。
9. reference 指示对函数或过程的引用。
10. operator 运算符指示类运算符。
11. platform, deprecated, experimental, 和 library 是提示（或警告）指令。这些指令在编译时生成警告。
12. assembler 和 export 指令没有任何意义。它们的存在只 是为了向后兼容。

指令字类型

Delphi 有两种类型的指令，包括上面指令表中列出的上下文相关类型的指令。

上下文相关指令可以是标识符（通常不是关键字），放置在声明的末尾以修改声明的含义。例如：

```
  procedure P; forward;
Or:
   procedure M; virtual; override;
Or:
   property Foo: Integer read FFoo write FFoo default 42;
```

最后一种类型的指令是正式的编译器指令，它是影响编译器行为的开关或选项。编译器指令用大括号 （{}） 括起来，并以美元符号 （$） 开头，如下所示：

```
{$POINTERMATH ON}

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

与其他类型的指令一样，编译器指令不是关键字。有关编译器指令的列表，请参阅 Delphi 编译器指令列表。

数字值

整数和实数常量可以用十进制表示法表示为不带逗号或空格的数字序列，并以 + 或 - 运算符为前缀以表示符号。值默认为正（例如，67258 等效于 +67258），并且必须在最大预定义实数或整数类型的范围内。

带小数点或指数的数字表示实数，而其他数字表示整数。当字符 E 或 e 出现在实数中时，它的意思是“乘以十的幂”。例如，7E2 表示 7 * 10^2,12.25e+6 和 12.25e6 都表示 12.25 * 10^6。

美元符号前缀表示十六进制数字，例如 $8F。没有前一元运算符的十六进制数被视为正值。在分配期间，如果十六进制值超出接收类型的范围，则会引发错误，但引发警告的整数（32 位整数）除外。在这种情况下，超过 Integer 正范围的值以与 2 的补码整数表示一致的方式被

视为负数。

有关实数和整数类型的详细信息，请参阅关于数据类型 （Delphi）。有关数字的数据类型的信息，请参阅声明的常量。

标签

可以使用标识符或非负整数作为标签。Delphi 编译器允许从 0 到 4294967295（uint32 范围）的数字标签。

标签用于 goto 语句。有关 goto 语句和标签的详细信息，请参阅声明和语句中的 Goto 语句 （Delphi）。

字符串

字符串（也称为字符串文本或字符串常量）由带引号的字符串、控制字符串或带引号的字符串和控制字符串的组合组成。分隔符只能出现在带引号的字符串中。

带引号的字符串是来自 ANSI 或多字节字符集的字符序列，写在一行上并用引号括起来。引号之间空白的字符串是空字符串。带引号的字符串中的两个连续引号表示单个字符，即引号。

该字符串在内部表示为编码为 UTF-16 的 Unicode 字符串。基本多语言平面 （BMP） 中的字符需要 2 个字节，BMP 中没有的字符需要 4 个字节。

例如：

```
'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 }
```

控制字符串是由一个或多个控制字符组成的序列，每个控制字符由 # 符号后跟一个无符号整数常量组成，该常量从 0 到 65,535（十进制）或从 $0 到 $FFFF（十六进制），采用 UTF-16 编码，并表示对应于指定代码值的字符。每个整数在内部由字符串中的 2 个字节表示。这对于表示控制字符和多字节字符很有用。控制字符串：

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

等效于带引号的字符串：

```
'You'
```

可以将带引号的字符串与控制字符串组合在一起，以形成更大的字符串。例如，您可以使用：

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

在“Line 1”和“Line 2”之间放置回车换行符。但是，不能以这种方式连接两个带引号的字符串，因为一对连续的撇号被解释为单个字符。（若要连接带引号的字符串，请使用 + 运算符或简单地将它们组合成单个带引号的字符串。

字符串与任何字符串类型和 PChar 类型兼容。由于 AnsiString 类型可能包含多字节字符，因此具有一个字符（单字节或多字节）的字符串与任何字符类型兼容。启用扩展语法（使用编译器指令 {$X+}）时，长度为 n 的非空字符串与从零开始的数组和 n 个字符的打包数组兼容。有关详细信息，请参阅关于数据类型 （Delphi）。

注释和编译器指令

编译器将忽略注释，除非它们充当分隔符（分隔相邻标记）或编译器指令。

有几种方法可以构造注释：

```
{ 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.
```

相似的注释不能嵌套。例如，（*{}*） 将。后一种形式对于注释掉也包含注释的代码部分很有用。

以下是有关如何以及何时使用三种类型的注释字符的一些建议：

- ✓ 使用双斜杠（//）注释掉开发期间所做的临时更改。您可以使用代码编辑器方便的 CTRL+/（斜杠）机制在工作时快速插入双斜杠注释字符。
- ✓ 使用括号星号“（*...*）”既用于开发注释，也用于注释掉包含其他注释的代码块。此注释字符允许编译器从考虑中删除多行源代码，包括其他类型的注释。
- ✓ 对要保留代码的源内文档使用大括号 （{}）。

紧跟在开头 { 或 （* 之后的包含美元符号 （$） 的注释是编译器指令。例如

```
{$WARNINGS OFF}
```

告知编译器不要生成警告消息。

in 保留字

- ✓ 检查集合中是否存在项目

```
A in Set1
```

- ✓ 循环访问容器的项

```
for Element in ArrayExpr do Stmt;
```

表达式

表达式

表达式是返回值的构造。下表显示了 Delphi 表达式的示例：

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

最简单的表达式是变量和常量（在关于数据类型 （Delphi）中描述）。更复杂的表达式是使用运算符、函数调用、集合构造函数、索引和类型转换从更简单的表达式构建的。

运算符

运算符的行为类似于作为 Dephi 语言一部分的预定义函数。例如，表达式 （X + Y） 由变量 X 和 Y（称为操作数）构建，带有 + 运算符;当 X 和 Y 表示整数或实数时，（X + Y） 返回它们的总和。运算符包括 @、not、^、*、/、div、mod 和 shl、shr、as、+、-、or、xor、=、>、<、<>、<=、>=、in 和 is。

运算符 @、not 和 ^ 是一元的（取一个操作数）。所有其他运算符都是二元的（采用两个操作数），除了 + 和 - 可以用作一元或二元运算符。一元运算符始终位于其操作数之前（例如 -B），但 ^ 除外，^ 位于其操作数之后（例如 P^）。二元运算符位于其操作数之间（例如，A = 7）。

某些运算符的行为会有所不同，具体取决于传递给它们的数据类型。例如，not 对整数操作数执行按位求反，对布尔操作数执行逻辑否定。此类运算符在下面出现在多个分类下。

除了 ^、is 和 in 之外，所有运算符都可以采用 Variant 类型的操作数;有关详细信息，请参阅变体类型 （Delphi）。

以下各节假定对 Delphi 数据类型有一定的了解;有关详细信息，请参阅关于数据类型（Delphi）。

有关复杂表达式中的运算符优先级的信息，请参阅本主题后面的运算符优先级规则。

算数运算符

采用实数或整数操作数的算术运算符包括 +、-、*、/、div 和 mod。

二元算数运算符

运算符	操作含义	支持数据类型	结果类型	示例
+	加法	integer, real	integer, real	X + Y
-	减法	integer, real	integer, real	Result -1
*	乘法	integer, real	integer, real	P * InterestRate
/	小数除法	integer, real	real	X / 2
div	整数除法	integer	integer	Total div UnitSize
mod	求余数	integer	integer	Y mod 6

一元算数运算符

运算符	操作含义	支持数据类型	结果类型	示例
+	正数	integer, real	integer, real	+7
-	负数	integer, real	integer, real	-X

以下规则适用于算术运算符：

- ✓ x / y 的值为 Extended 类型，与 x 和 y 的类型无关。对于其他算术运算符，只要至少有一个操作数为实数，结果的类型为 Extended ;否则，当至少一个操作数为 Int64 类型时，结果为 Int64 类型;否则，结果的类型为 Integer。如果操作数的类型是 Integer 类型的子范围，则将其视为整数类型。
- ✓ x div y 的值是 x / y 的值，沿零方向舍入到最接近的整数。
- ✓ mod 运算符返回通过除其操作数获得的余数。换句话说，x mod y = x - （x div y） * y.
- ✓ 当 y 在形式为 x / y、x div y 或 x mod y 的表达式中为零时，会发生运行时错误。

布尔运算符

布尔运算符 not、and、or 和 xor 接受任何布尔类型的操作数，并返回布尔类型的值。

运算符	操作含义	支持数据类型	结果类型	示例
not	非	Boolean	Boolean	not (C in MySet)
and	且	Boolean	Boolean	Done and (Total >0)
or	或	Boolean	Boolean	A or B
xor	异或	Boolean	Boolean	A xor B

这些操作由布尔逻辑的标准规则控制。例如，当且仅当 x 和 y 均为 True 时，形式为 x 和 y 的表达式为 True。

依次求值和短路逻辑与

编译器支持 and 和 or 运算符的两种计算模式：完全计算和短路（部分）计算。完全求值意味着即使已经确定了整个表达式的结果，也会评估每个求值。短路评估意味着严格的从左到右评估，一旦确定整个表达式的结果，就会停止。例如，如果在短路模式下计算表达式 A and B，当 A 为 False 时，编译器不会计算 B;它一计算 A 就知道整个表达式为 False。

短路评估通常是可取的，因为它保证了最短的执行时间，并且在大多数情况下保证了最小的代码大小。当一个操作数是一个具有改变程序执行的副作用的函数时，完整的计算有时很方便。

短路评估还允许使用可能导致非法运行时操作的结构。例如，以下代码循环访问字符串 S，直到第一个逗号。

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

在 S 没有逗号的情况下，最后一次迭代将 I 递增到大于 S 长度的值。下次测试 while 条件时，完整的计算结果是尝试读取 S[I]，这可能会导致运行时错误。相反，在短路评估下，while 条件的第二部分 （S[I] <> ''） 在第一部分发生故障后不被评估。

使用 $B 编译器指令控制计算模式。默认状态为 {$B}，启用短路评估。若要在本地启用完整计算，请将 {$B+} 指令添加到代码中。还可以通过在“编译器选项”对话框中选择“完成布尔计算”，在项目范围内切换到完成计算（所有源单元都需要重新编译）。

注意：如果任一操作数涉及变体，编译器始终执行完整计算（即使在 {$B} 状态下也是如此。

逻辑（位）运算符

以下逻辑运算符对整数操作数执行按位操作。例如，如果存储在 X（二进制）中的值 001101，存储在 Y 中的值 100001，则语句：

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

将值 101101 分配给 Z。

运算符	操作含义	支持数据类型	结果类型	示例
not	非	integer	integer	not X
and	与	integer	integer	X and Y
or	或	integer	integer	X or Y
xor	异或	integer	integer	X xor Y
shl	无符号左移	integer	integer	X shl 2
shr	无符号右移	integer	integer	Y shr I

以下规则适用于按位运算符：

- ✓ not 操作的结果与操作数的类型相同。
- ✓ 如果 and、or 或 xor 运算的操作数都是整数，则结果是预定义的整数类型，其最小范围包括这两种类型的所有可能值。
- ✓ 运算 x shl y 和 x shr y 将 x 的值向左或向右移动 y 位，这（如果 x 是无符号整数）相当于将 x 乘以或除以 2^y;结果与 X 的类型相同。例如，如果 N 存储值 01101（十进制 13），则 N shl 1 返回 11010（十进制 26）。请注意，y 的值被解释为 x 类型的大小的模数。因此，例如，如果 x 是一个整数，则 x shl 40 被解释为 x shl 8，因为整数是 32 位，40 mod 32 是 8。

如果 x 是负整数，则在以下示例中明确了 shl 和 shr 操作：

```
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.
```

字符串运算符

关系运算符 =、<>、<、>、<= 和 >= 都采用字符串操作数（请参阅本节后面的关系运算符）。+ 运算符连接两个字符串。

运算符	操作含义	支持数据类型	结果类型	示例
+	链接字符串	string, packed string, character	string	S + '.'

以下规则适用于字符串串联：

- ✓ + 的操作数可以是字符串、 packed string（Char 类型的打包数组）或字符。但是，如果一个操作数的类型为 WideChar，则另一个操作数必须是长字符串（UnicodeString、AnsiString 或 WideString）。
- ✓ + 运算的结果与任何字符串类型兼容。但是，如果操作数都是短字符串或字符，并且其组

合长度大于 255，则结果将被截断为前 255 个字符。

指针运算符

关系运算符 <、>、<= 和 >= 可以采用 PAnsiChar 和 PWideChar 类型的操作数（请参阅关系运算符）。以下运算符也将指针作为操作数。有关指针的详细信息，请参阅关于数据类型（Delphi） 中的指针和指针类型 （Delphi）。

字符串指针运算符

运算符	操作含义	支持数据类型	结果类型	示例
+	指针加法	character pointer, integer	character pointer	P + I
-	指针减法	character pointer, integer	character pointer, integer	P - Q
^	指针解引用	pointer	base type of pointer	P^
=	指针相等判断	pointer	Boolean	P = Q
<>	指针不等判断	pointer	Boolean	P <> Q

^ 运算符取消引用指针。其操作数可以是除泛型指针之外的任何类型的指针，泛型指针必须在取消引用之前进行类型转换。

P = Q 为真，以防 P 和 Q 指向同一地址;否则，P <> Q 为真。

可以使用 + 和 - 运算符来递增和减少字符指针的偏移量。您还可以使用 - 来计算两个字符指针的偏移量之间的差异。以下规则适用：

如果 I 是整数，P 是字符指针，则 P + I 将 I 添加到 P 给出的地址;也就是说，它返回一个指向 P 后面的地址 I 字符的指针（表达式 I + P 等效于 P + I。P - I 从 P 给出的地址中减去 I;也就是说，它返回一个指向 P 之前的地址 I 字符的指针。对于 PAnsiChar 指针也是如此;对于 PWideChar 指针 P + I 将 I * SizeOf（WideChar） 添加到 P。

如果 P 和 Q 都是字符指针，则 P - Q 计算 P 给出的地址（较高的地址）和 Q 给出的地址（较低的地址）之间的差;也就是说，它返回一个整数，表示 P 和 Q 之间的字符数。

P + Q 未定义。

集合运算符

以下运算符将集合作为操作数。

运算符	操作含义	支持数据类型	结果类型	示例
+	并集	set	set	Set1 + Set2
-	集合减法	set	set	S - T
*	交集	set	set	S * T
<=	子集判断	set	Boolean	Q <= MySet
>=	超集判断	set	Boolean	S1 >= S2
=	相等判断	set	Boolean	S2 = MySet

<>	不等判断	set	Boolean	MySet <> S1
in	成员检测	ordinal, set	Boolean	A in Set1

以下规则适用于 +、- 和 *：

- ✓ 序数 O 在 X + Y 中当且仅当 O 在 X 或 Y（或两者）中。O 在 X - Y 中当且仅当 O 在 X 中但不在 Y 中。 O 在 X * Y 中，当且仅当 O 同时在 X 和 Y 中。
- ✓ +、- 或 * 运算的结果是 A 的类型集。B，其中 A 是结果集中最小的序号值，B 是最大的序号值。

以下规则适用于 <=、>=、=、<> 和 in：

- ✓ X <= Y 为真，X 的每个成员都是 Y 的成员;Z >= W 等价于 W <= Z.U = V 为 True，U 和 V 包含完全相同的成员;否则，U <> V 为真。
- ✓ 对于序数 O 和集合 S，O in S 为真，则 O 是 S 的成员。

关系运算符

关系运算符用于比较两个操作数。运算符 =、<>、<= 和 >= 也适用于集合。

运算符	操作含义	支持数据类型	结果类型	示例
=	相等	simple, class, class reference, interface, string, packed string	Boolean	✓ I = Max
<>	不相等	simple, class, class reference, interface, string, packed string	Boolean	X <> Y
<	小于	simple, string, packed string, PChar	Boolean	X < Y
>	大于	simple, string, packed string, PChar	Boolean	Len > 0
<=	小于等于	simple, string, packed string, PChar	Boolean	Cnt <= I
>=	大于等于	simple, string, packed string, PChar	Boolean	I >= 1

对于大多数简单类型，比较很简单。例如，I = J 为真，I 和 J 具有相同的值，否则 I <> J 为真。以下规则适用于关系运算符。

- ✓ 操作数必须是兼容的类型，但实数和整数可以比较。
- ✓ 根据构成字符串的字符的序号值比较字符串。字符类型被视为长度为 1 的字符串。
- ✓ 两个打包字符串必须具有相同数量的组件才能进行比较。将包含 n 个组件的打包字符串与字符串进行比较时，打包字符串被视为长度为 n 的字符串。
- ✓ 仅当两个指针指向同一字符数组中时，才使用运算符 <、>、<= 和 >= 来比较 PAnsiChar（和 PWideChar）操作数。
- ✓ 运算符 = 和 <> 可以采用类和类引用类型的操作数。对于类类型的操作数，= 和 <> 根据适用于指针的规则进行评估：C = D 为 True，C 和 D 指向同一实例对象，否则 C <> D 为 True。对于类引用类型的操作数，C = D 在 C 和 D 表示同一类的情况下为 True，否则 C

<> D 为 True。这不会比较存储在类中的数据。有关类的详细信息，请参阅类和对象（Delphi）。

类和接口运算符

运算符 as 和 is 将类和实例对象作为操作数;AS 也在接口上运行。有关更多信息，请参见类和对象 （Delphi）、对象接口 （Delphi） 和接口引用 （Delphi）。

关系运算符 = 和 <> 也对类进行操作。

@运算符

@ 运算符返回变量或函数、过程或方法的地址;也就是说，@ 构造一个指向其操作数的指针。有关指针的详细信息，请参阅关于数据类型 （Delphi） 中的“指针和指针类型”。以下规则适用于 @。

如果 X 是变量，则@X 返回 X 的地址（当 X 是过程变量时，特殊规则适用;请参阅关于数据类型 （Delphi） 中的“语句和表达式中的过程类型”。如果默认 {$T} 编译器指令有效，则@X 的类型为 指针。在 {$T+} 状态下，@X 的类型为 ^T，其中 T 是 X 的类型（此区别对于赋值兼容性很重要，请参阅赋值兼容性）。

如果 F 是例程（函数或过程），则@F 返回 F 的入口点。@F 的类型始终为指针。

将 @ 应用于类中定义的方法时，必须使用类名限定方法标识符。例如

```
@TMyClass.DoSomething
```

指向 TMyClass 的 DoSomething 方法。有关类和方法的详细信息，请参阅类和对象（Delphi）。

注意：使用 @ 运算符时，无法获取接口方法的地址，因为该地址在编译时未知，无法在运行时提取。

运算符优先级

在复杂表达式中，优先级规则确定执行操作的顺序。

运算符	优先级（从高到低）
@ not	first (最高优先级)
* / div mod and shl shr	second

as	
+ - or xor	third
= <> < > <= >= in is	fourth (最低优先级)

具有较高优先级的运算符先于具有较低优先级的运算符进行计算，而具有相同优先级的运算符则与左侧相关联。因此，表达式：

X + Y * Z
将 Y 乘以 Z，然后将 X 加到结果中;* 首先执行，因为 的优先级高于 +。但：

X - Y + Z
首先从 X 中减去 Y，然后将 Z 加到结果中;- 和 + 具有相同的优先级，因此首先执行左侧的操作。

您可以使用括号来覆盖这些优先规则。首先计算括号内的表达式，然后将其视为单个操作数。例如：

（X + Y） * Z
将 Z 乘以 X 和 Y 的总和。

有时在乍一看似乎不需要括号的情况下需要括号。例如，考虑以下表达式：

X = Y or X = Z
对此的预期解释显然是：

（X = Y） or （X = Z）
但是，如果没有括号，编译器遵循运算符优先级规则并将其读取为：

（X = （Y or X）） = Z
这会导致编译错误，除非 Z 是布尔值。

括号通常使代码更易于编写和阅读，即使严格来说它们是多余的。因此，第一个示例可以写成：

X + （Y * Z）

这里的括号是不必要的（对编译器来说），但它们使程序员和读者都不必考虑运算符优先级。

函数调用

由于函数返回值，因此函数调用是表达式。例如，如果您定义了一个名为 Calc 的函数，该函数接受两个整数参数并返回一个整数，则调用 Calc（24,47） 的函数是一个整数表达式。如果 I 和 J 是整数变量，那么 I + Calc（J，8） 也是一个整数表达式。函数调用的示例包括：

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

集合构建

集合构造函数表示集合类型值。例如：

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

表示其成员为 5、6、7 和 8 的集合。集合构造函数：

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

也可以表示相同的集合。

集合构造函数的语法为：

```
[ 项目 1， ...， 项目 n ]
```

其中每个项要么是一个表达式，表示集合基类型的序号，要么是一对中间有两个点 （..）的此类表达式。当项目具有 x.y，它是从 x 到 y 范围内所有序数的简写，包括 y;但如果 x 大于 y，则 x..y，集合 [x..y]，表示任何内容，并且是空集。集合构造函数 [ ] 表示空集合，而 [x] 表示其唯一成员是 x 值的集合。

集合构造函数的示例：

```
[红色，绿色，我的彩色]
[1， 5， 10..K mod 12， 23]
[“A..”Z'， 'a'..”z'， Chr（数字 + 48）]
```

有关集合的详细信息，请参阅关于数据类型 （Delphi） 中的结构化类型 （Delphi）。

索引访问

可以对字符串、数组、数组属性以及指向字符串或数组的指针编制索引。例如，如果 FileName 是一个字符串变量，则表达式 FileName[3] 返回字符串中由 FileName 表示的第三个字符，而 FileName[I + 1] 返回紧跟在由 I 编制索引的字符之后的字符。有关字符串的信息，请参阅数据类型、变量和常量。有关数组和数组属性的信息，请参阅数据类型、变量和常量中的数组和属性 （Delphi） 页中的“数组属性”。

类型转换

有时，将表达式视为属于不同类型的表达式很有用。类型转换实际上允许您通过临时更改表达式的类型来执行此操作。例如，整数('A')将字符 A 强制转换为整数。

类型转换的语法为：

类型标识符（表达式）
如果表达式是变量，则结果称为变量类型转换;否则，结果为值类型转换。虽然它们的语法相同，但不同的规则适用于这两种类型转换。

值类型转换

在值类型转换中，类型标识符和强制转换表达式都必须是序号类型或指针类型。值类型转换的示例包括：

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

结果值是通过转换括号中的表达式获得的。如果指定类型的大小与表达式的大小不同，这可能涉及截断或扩展。表达式的符号始终保留。

以下语句：

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

将 Integer('A') 的值 65 分配给变量 I。

值类型转换不能后跟限定符，也不能显示在赋值语句的左侧。

变量类型转换

您可以将任何变量转换为任何类型，前提是它们的大小相同，并且您不将整数与实数混合。（要转换数值类型，请依赖 Int 和 Trumc 等标准函数。变量类型转换的示例包括：

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

变量类型转换可以出现在赋值语句的任一侧。因此：

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

将字符 z （ASCII 122） 分配给 MyChar。

可以将变量强制转换为过程类型。例如，给定声明：

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

您可以进行以下赋值语句：

```
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 }
```

变量类型转换后还可以跟限定符，如以下示例所示：

```
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;
```

在此示例中，TByteRec 用于访问单词的低阶和高阶字节，TWordRec 用于访问长整数的低阶和高阶单词。您可以出于相同的目的调用预定义函数 Lo 和 Hi，但变量类型转换的优点是它可以在赋值语句的左侧使用。

有关类型转换指针的信息，请参阅指针和指针类型 （Delphi）。有关转换类和接口类型的信息，请参阅类引用和接口引用 （Delphi） 中的“As 运算符”。

声明和语句

本主题介绍 Delphi 声明和语句的语法。

除了 uses 子句（以及像实现这样的保留字，它划分了单元的各个部分）之外，程序完全由声明和语句组成，这些声明和语句被组织成块。

本主题涵盖以下项目：

- ✓ 声明
- ✓ 简单的语句，例如赋值
- ✓ 结构化语句，例如条件测试（例如，if-then 和 case）、迭代（例如，for 和 while）。

声明

变量、常量、类型、字段、属性、过程、函数、程序、单元、库和包的名称称为标识符（像 26057 这样的数字常量不是标识符）。必须先声明标识符，然后才能使用它们;唯一的例外是编译器自动理解的一些预定义类型、例程和常量、变量 Result （当它出现在函数块内）和变量 Self 当它出现在类的方法实现中时。

声明定义标识符，并在适当的情况下为其分配内存。例如：

```
var Size: Extended;
```

声明一个名为 Size 的变量，该变量保存扩展（实数）值，同时：

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

声明一个名为 DoThis 的函数，该函数将两个字符串作为参数并返回一个整数。每个声明都以分号结尾。同时声明多个变量、常量、类型或标签时，只需编写一次相应的保留字：

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

声明的语法和位置取决于您定义的标识符类型。通常，声明只能出现在块的开头或单元的接口或实现部分的开头（在 uses 子句之后）。这些主题的文档介绍了用于声明变量、常量、类型、函数等的特定约定。

提示指令

“提示”指令 platform, deprecated, 和 library 可以附加到任何声明中。这些指令将在编译时生成警告。提示指令可以应用于类型声明、变量声明、类、接口和结构声明、类或记录中的字段声明、过程、函数和方法声明以及单元声明。

当提示指令出现在单元声明中时，表示该提示适用于单元中的所有内容。例如，Windows 上的 Windows 3.1 样式 OleAuto.pas 单元已完全弃用。对该单元或该单元中的任何符号的任何引用都会生成弃用警告消息。

符号或单元上的 platform 提示指令表明它可能不存在，或者实现在不同平台上可能会有很大差异。符号或单元上的 library 提示指令指示代码可能不存在，或者实现在不同的库体系结构上可能有很大差异。

platform 和 library 指令不指定哪个平台或库。如果您的目标是编写独立于平台的代码，则无需知道符号特定于哪个平台;将符号标记为特定于某个平台就足够了，让您知道它可能会给您的可移植性目标带来问题。

对于过程或函数声明，提示指令应用分号与声明的其余部分分开。例子：

```
procedure SomeOldRoutine; stdcall deprecated;

 var
    VersionNumber: Real library;

 type
    AppError = class(Exception)
        ...
```

end platform;

在 {$HINTS ON} {$WARNINGS ON} 状态下编译源代码时，对使用这些指令之一声明的标识符的每个引用都会生成相应的提示或警告。使用平台标记特定于特定操作环境（如 Windows）的项，已弃用以指示项已过时或仅支持向后兼容，并使用库标记对特定库或组件框架的依赖项。

Delphi 编译器还识别 experimental 提示指令。您可以使用此指令指定处于不稳定开发状态的单元。编译器在生成使用该单元的应用程序时将发出警告。

有关 Delphi 提示指令的详细信息，请参阅方法声明中的警告指令。所有 Delphi 指令都列在指令中。

语句

语句定义程序中的算法操作。简单的语句（如赋值和过程调用）可以组合成循环、条件语句和其他结构化语句。

块内以及单元的初始化或销毁部分中的多个语句用分号分隔。

简单语句

简单语句不包含任何其他语句。简单语句包括赋值、对过程和函数的调用以及 goto 跳转。

赋值语句

赋值语句的格式为：

variable := expression

其中变量是任何变量引用，包括变量、变量类型转换、取消引用指针或结构化变量的组件。表达式是任何与赋值兼容的表达式（在功能块中，变量可以替换为正在定义的函数的名称。请参阅过程和函数 （Delphi）。 := 符号有时称为赋值运算符。

赋值语句将变量的当前值替换为表达式的值。例如：

I := 3;

将值 3 赋给变量 I。赋值左侧的变量引用可以显示在右侧的表达式中。例如：

I := I + 1;

递增 I 的值。其他赋值语句包括：

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;
```

过程和函数调用语句

过程调用由过程的名称（带或不带限定符）组成，后跟参数列表（如果需要）。示例包括：

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

启用扩展语法 （{$X+}） 后，函数调用（如对过程的调用）本身可以被视为语句：

```
MyFunction(X);
```

以这种方式使用函数调用时，将丢弃其返回值。
有关过程和函数的详细信息，请参阅过程和函数 （Delphi）。

Goto 语句

goto 语句，其形式为：

```
goto label
```

将程序执行传输到由指定标签标记的语句。若要标记语句，必须先声明标签。然后，必须在要标记的语句前面加上标签和冒号：

```
label: statement
```

声明如下标签：

```
label label;
```

您可以一次声明多个标签：

label label1, ..., labeln;

标签可以是任何有效的标识符或 0 到 4294967295 之间的任何数字。

标签声明、标记语句和 goto 语句必须属于同一块。（请参阅下面的块和范围。因此，不可能跳入或跳出过程或函数。不要在具有相同标签的块中标记多个语句。

例如：

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

创建一个重复调用 Beep 过程的无限循环。

此外，无法跳入或退出 try - finally 或 try - except 语句。

在结构化编程中通常不鼓励使用 goto 语句。但是，它有时用作退出嵌套循环的一种方式，如以下示例所示：

```
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;
```

请注意，我们正在使用 goto 跳出嵌套循环。切勿跳入循环或其他结构化语句，因为这会产生不可预测的效果。

结构化语句

复合语句是按编写顺序执行的其他（简单或结构化）语句的序列。复合语句由保留字 begin 和 end 括起来，其组成语句用分号分隔。例如：

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

```
  end;
```

结束前的最后一个分号是可选的。所以这可以写成：

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

复合语句在 Delphi 语法需要单个语句的上下文中是必不可少的。除了程序、函数和过程块之外，它们还出现在其他结构化语句中，例如条件或循环。例如：

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

您可以编写仅包含一个构成语句的复合语句;就像复杂术语中的括号一样，begin 和 end 有时用于消除歧义并提高可读性。还可以使用空的复合语句来创建不执行任何操作的块：

```
begin
end;
```

with 语句

with 语句是引用记录字段或对象的字段、属性和方法的简写。with 语句的语法为：

```
with obj do statement
```

or:

```
with obj1, ..., objn do statement
```

其中 obj 是生成对记录、对象实例、类实例、接口或类类型（元类）实例的引用的表达式，语句是任何简单或结构化语句。在语句中，您可以单独使用字段、属性和 obj 的标识符来引用它们，即不使用限定符。

例如，给定声明：

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

 var
    OrderDate: TDate;
```

您可以使用 with 语句编写以下代码：

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

或者，您可以在不使用 with 语句的情况下编写以下代码：

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

如果 obj 的解释涉及索引数组或取消引用指针，则在执行语句之前执行一次这些操作。这使得语句高效而简洁。这也意味着在当前执行 with 语句期间，对语句中变量的赋值不能影响对 obj 的解释。

如果可能，with 语句中的每个变量引用或方法名称都将解释为指定对象或记录的成员。如果要从 with 语句访问另一个同名的变量或方法，则需要在它前面加上限定符，如以下示例所示：

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

当 with 之后出现多个对象或记录时，整个语句被视为一系列嵌套的语句。因此：

```
with obj1, obj2, ..., objn do statement
is equivalent to:

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

在这种情况下，如果可能，语句中的每个变量引用或方法名称都会被解释为 objn 的成员;否则，如果可能的话，它被解释为 OBJN1 的成员;等等。同样的规则也适用于解释 objs 本身，因此，例如，如果 objn 同时是 obj1 和 obj2 的成员，则将其解释为 obj2.objn。

由于 with 语句需要对变量或字段进行操作，因此将其与属性一起使用有时会很棘手。with 语句期望它所操作的变量可以通过引用获得。

使用时要注意的最重要的事项：

- ✓ 只能将 with 用于只读属性进行读取。尝试修改公开的记录或对象中的字段会导致编译时错误。
- ✓ 即使该属性允许对字段进行写入访问，您仍然无法使用 来修改其字段。

下面的代码演示了在公开记录的只读属性上使用 with 语句时出现的问题。假设您有以下类：

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

其中 TPoint 是声明如下的记录：

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

通常，Center 属性是只读的，不允许您修改 FCenter 字段的值或字段。在这种情况下，使用如下所示的 with 语句将失败并出现编译时错误，因为 Shape.Center 不是变量，并且您不能引用它：

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

```
    Y := 100;
  end;
```

使用 with 语句时棘手的部分来自读/写属性。我们更改了原始 TShape 类，以允许对其 FCenter 字段进行写入访问：

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

即使 Center 属性不是只读的，也会发出相同的编译时错误。此问题的解决方案是更改如下所示的代码：

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

改为如下所示的代码：

```
 { 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 语句

if 语句有两种形式：if...then 和 if...then...else。if...then 语句如下：

```
if expression then statement
```

其中表达式返回布尔值。如果表达式为 True，则执行语句;否则不执行。例如：

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

if...then...else 语句如下：

```
if expression then statement1 else statement2
```

其中表达式返回布尔值。如果表达式为 True，则执行语句 1;否则执行语句 2。例如：

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

then 和 else 子句各包含一个语句，但它可以是结构化语句。例如：

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

请注意，从不使用 then 子句和单词 else 之间的分号。您可以在整个 if 语句之后放置一个分号，以将其与其块中的下一条语句分开，但 then 和 else 子句只需要它们之间的空格或回车符即可。将分号放在紧靠其他字符之前（在 if 语句中）是一个常见的编程错误。

嵌套的 if 语句会出现一个特殊的困难。发生这种情况是因为某些 if 语句具有 else 子句，而另一些语句没有，但这两种语句的语法在其他方面是相同的。在一系列嵌套条件中，else 子句比 if 语句少，似乎不清楚哪些 else 子句绑定到哪个 if。考虑以下形式的语句：

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

似乎有两种方法可以解析它：

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

但是，编译器始终以第一种方式进行分析。也就是说，在实际代码中，语句：

```
 if ... { expression1} then
    if ... {expression2} then
```

```
        ... {statement1}
    else
        ... {statement2}
```

相当于:

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

规则是嵌套条件从最里面的条件开始解析，如果每个条件在其左侧，则每个条件绑定到最近的可用条件。要强制编译器以第二种方式读取我们的示例，您必须将其显式编写为：

```
 if ...
{expression1}

then
   begin
      if ...
{expression2}

then
         ...
{statement1}

   end
   else
      ...
{statement2}

;
```

case 语句

case 语句可以为深度嵌套的 if 条件提供可读的替代方法。case 语句的格式为：

```
 case selectorExpression of
    caseList1: statement1;
     ...
    caseListn: statementn;
```

```
end
```

其中 selectorExpression 是小于 32 位的序号类型的任何表达式（字符串类型和大于 32 位的序数无效），每个 caseList 是以下之一：

- ✓ 编译器无需执行程序即可计算的数字、声明的常量或其他表达式。它必须是与选择器表达式兼容的序号类型。因此，7、True、4 + 5 * 3、'A' 和 Integer('A') 都可以用作 caseLists，但变量和大多数函数调用不能。（一些内置函数（如 Hi 和 Lo）可能会出现在案例列表中。请参阅声明的常量。
- ✓ First..Last 子范围。其中“First”和“Last”都满足上述条件，“First”小于或等于“Last”。
- ✓ 具有 item1， ...， itemn 形式的列表，其中每个项目都满足上述条件之一。
- ✓ case 列表表示的每个值在案例语句中必须是唯一的;子范围和列表不能重叠。
- ✓ case 语句可以有一个最终的 else 子句：

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

其中语句是以分号分隔的语句序列。当执行 case 语句时，最多一个语句 1 ...语句 n 被执行。无论哪个 caseList 的值等于 selectorExpression 的值，都会确定要使用的语句。如果 caseList 中没有一个与 selectorExpression 具有相同的值，则执行 else 子句中的语句（如果有的话）。

以下 case 语句：

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

等效于嵌套条件：

```
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 := '';
```

其它示例 case 语句：

```
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;
```

循环控制语句

循环允许您重复执行一系列语句，使用控制条件或变量来确定执行何时停止。Delphi 有三种控制循环：repeat 语句、while 语句和 for 语句。

可以使用标准的“break”和“continue”过程来控制 repeat、while 或 for 语句的流。break 终止发生它的语句，而 continue 开始执行下一次迭代。

repeat 语句

repeat 语句语法如下：

```
repeat statement1; ...; statementn; until expression
```

其中表达式返回布尔值。（until 之前的最后一个分号是可选的。repeat 语句连续执行其组成语句序列，在每次迭代后测试表达式。当表达式返回 True 时，repeat 语句终止。该序列始终至少执行一次，因为表达式直到第一次迭代之后才会计算。

repeat 语句的示例包括：

```
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 语句

while 语句类似于 repeat 语句，不同之处在于控制条件是在语句序列的第一次执行之前计算的。因此，如果条件为 false，则永远不会执行语句序列。

while 语句的语法为：

```
while expression do statement
```

其中表达式返回布尔值，语句可以是复合语句。while 语句重复执行其构成语句，在每次迭代之前测试表达式。只要表达式返回 True，执行就会继续。

while 语句的示例包括：

```
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 语句

与 repeat 或 while 语句不同，for 语句要求您显式指定希望循环经历的迭代次数。for 语句的语法为：

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

或:

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

其中：

- ✓ counter 是序号类型的局部变量（在包含 for 语句的块中声明），没有任何限定符。
- ✓ initialValue 和 finalValue 是与计数器赋值兼容的表达式。
- ✓ 语句是不更改 counter 值的简单或结构化语句。

for 语句将 initialValue 的值分配给 counter，然后重复执行语句，每次迭代后递增或递减 counter。（for...to 语法增量 counter，而 for ...downto 语法递减 counter。当 counter 返回与 finalValue 相同的值时，将再次执行语句，并且 for 语句终止。换句话说，语句对从 initialValue 到 finalValue 范围内的每个值执行一次。如果 initialValue 等于 finalValue，则语句只执行一次。如果 initialValue 大于 for...to 语句中 finalValue ，或小于 for...downto 语句中 finalValue ，则语句永远不会执行。for 语句终止后（前提是这不是由 Break 或 Exit 过程强制的），counter 的值是未定义的。

警告：迭代变量计数器不能在循环中修改。这包括赋值和将变量传递给过程的 var 参数。这样做会导致编译时警告。

为了控制循环的执行，表达式 initialValue 和 finalValue 在循环开始之前只计算一次。因此，对于 for...to 语句几乎（但不完全）等同于以下 while 语句：

```
begin
    counter := initialValue;
    while counter <= finalValue do
    begin
       ... {statement};
       counter := Succ(counter);
    end;
 end.
```

这种结构和 for...to 语句不同点在于 while 循环在每次迭代之前重新计算 finalValue。如果 finalValue 是一个复杂的表达式，这可能会导致性能明显降低，这也意味着对语句中 finalValue 值的更改可能会影响循环的执行。

示例语句：

```
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);
```

使用 for 语句迭代容器

Delphi 支持容器上的 for-element-in-collection 样式迭代。编译器可识别以下容器迭代模式：

- ✓ 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;

迭代变量 Element 的类型必须与容器中保存的类型匹配。对于循环的每次迭代，迭代变量都保存当前集合成员。与常规 for 循环一样，迭代变量必须在与 for 语句相同的块中声明。

警告：迭代变量不能在循环中修改。这包括赋值和将变量传递给过程的 var 参数。这样做会导致编译时警告。

数组表达式可以是单维或多维、固定长度或动态数组。数组按递增顺序遍历，从最低数组边界开始，到数组大小减一结束。以下代码显示了遍历单个、多维和动态数组的示例：

```
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.
```

下面的示例演示对字符串表达式的迭代：

```
  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.
```

下面的示例演示对集合表达式的迭代：

```
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.
```

若要在类或接口上使用 for-in 循环构造，该类或接口必须实现规定的集合模式。实现集合模式的类型必须具有以下属性：

✓ 类或接口必须包含名为 GetEnumerator（） 的公共实例方法。GetEnumerator（） 方法必

须返回类、接口或记录类型。

- ✓ GetEnumerator（） 返回的类、接口或记录必须包含一个名为 MoveNext（） 的公共实例方法。MoveNext（） 方法必须返回一个布尔值。for-in 循环首先调用此方法以确保容器不为空。
- ✓ GetEnumerator（） 返回的类、接口或记录必须包含名为 Current 的公共实例只读属性。Current 属性的类型必须是集合中包含的类型。

以下代码演示如何迭代 Delphi 中的可枚举容器。

```
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.
```

作为枚举器类型的一部分，RAD Studio 在迭代完成后正式定义枚举器状态：“在 MoveNext 返回 False 后，枚举器状态无效，必须释放或重新创建枚举器，不应进一步访问”。

使用 for 语句迭代数据集

Delphi 支持输入语法构造来迭代数据集。编译器识别以下数据集迭代模式：

```
for Record in Dataset do Smth;
```

其中记录由 TDataSet API 表示。可以安全地假设，记录等于数据集。

以下代码片段循环访问 Delphi 中的数据集。此示例代码说明如何将 Name 列的值输出到 Memo 控件。

```
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);
```

备注：数据集枚举不是可重新输入的操作。这意味着对于数据集，您目前只能使用一个枚举。如果需要同时对同一数据集执行多个 for-in 循环，请改用 TDataSet.View 方法（请参阅本主题后面的内容）。在这种情况下，在 for-in 循环中，记录可能不等于数据集。

下面的代码片段演示如何使用 TDataSet.View 方法枚举数据集。

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

支持迭代的内置类类型

以下类及其后代支持 for-in 语法：

- ✓ System.Classes.TList
- ✓ System.Classes.TCollection
- ✓ System.Classes.TStrings
- ✓ System.SysUtils.TStringBuilder
- ✓ 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

备注：当枚举器循环正在进行时，TStringBuilder 枚举器不支持 TStringBuilder 修改。

语句块和作用域

声明和语句被组织到定义标签和标识符的本地命名空间（或范围）的块中。块允许单个标识符（如变量名）在程序的不同部分中具有不同的含义。每个块都是程序、函数或过程声明的一部分;每个程序、函数或过程声明都有一个块。

语句块

块由一系列声明后跟复合语句组成。所有声明必须一起出现在块的开头。所以块的形式是：

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

声明部分可以按任意顺序包含变量、常量（包括资源字符串）、类型、过程、函数和标签的声明。在程序块中，声明部分还可以包含一个或多个导出子句（请参阅库和包 （Delphi））。

例如，在这样的函数声明中：

```
function UpperCase(const S: string): string;
  var
    Ch: Char;
```

```
    L: Integer;
    Source, Dest: PChar;
begin
    ...
end;
```

声明的第一行是函数标题，所有后续行组成块。Ch、L、Source 和 Dest 是局部变量;它们的声明仅适用于 UpperCase 函数语句块，并且仅在此块中覆盖程序块或单元的接口或实现部分中可能发生的相同标识符的任何声明。

作用域

标识符（如变量或函数名称）只能在其声明范围内使用。声明的位置决定了其范围。在程序、函数或过程的声明中声明的标识符的作用域仅限于声明它的块。在单元的接口部分中声明的标识符的作用域包括使用该单元进行声明的任何其他单元或程序。作用域较窄的标识符（尤其是在函数和过程中声明的标识符）有时称为本地标识符，而范围较宽的标识符称为全局标识符。

下面总结了确定标识符范围的规则。

声明位置	作用域.
程序、函数或过程的声明部分	从声明它的位置到当前块的末尾，包括该范围内包含的所有块。
单元文件的接口部分	从声明的点到单元的末尾，以及使用该单元的任何其他单元或程序。（请参阅程序和单元 （Delphi）。
单元的实现部分，但不在任何功能或过程的块内	从声明点到单元结束。标识符可用于单元中的任何函数或过程，包括初始化和完成部分（如果存在）。
记录类型中定义（即，标识符是记录中字段的名称）	从声明点到记录类型定义的末尾。（请参阅结构化类型（delphi）中的“记录”。
类中定义（即标识符是类中数据字段属性或方法的名称）	从其声明点到类类型定义的末尾，还包括类的后代和类中所有方法的块及其后代。（请参阅类和对象 （Delphi）。

名称冲突

当一个块包围另一个块时，前者称为外块，后者称为内块。如果在外部块中声明的标识符在内部块中重新声明，则内部声明优先于外部声明，并确定标识符在内部块持续时间内的含义。例如，如果在单元的接口部分中声明一个名为 MaxValue 的变量，然后在该单元内的函数声明中声明另一个同名的变量，则功能块中任何未限定出现的 MaxValue 都由第二个本地声明控制。同样，在另一个函数中声明的函数会创建一个新的内部作用域，在该作用域中，外部函数使用的标识符可以在本地重新声明。

使用多个单元使范围的定义进一步复杂化。uses 子句中列出的每个单元都施加了一个新

范围，该范围包含使用的其余单元以及包含 uses 子句的程序或单元。uses 子句中的第一个单元表示最外层的作用域，每个后续单元表示前一个作用域内的新作用域。如果两个或多个单元在其接口部分中声明相同的标识符，则对该标识符的非限定引用将在最内层的作用域（即引用本身所在的单元中）中选择声明，或者，如果该单元未声明标识符，则在声明标识符的 uses 子句中的最后一个单元中选择声明。

System 和 SysInit 单元由每个程序或单元自动使用。System 中的声明以及编译器自动理解的预定义类型、例程和常量始终具有最外层的作用域。

您可以使用限定标识符（请参阅基本语法元素 （Delphi） 中的“限定标识符”）或 with 语句（请参阅上面的“with 语句”）覆盖这些范围规则并绕过内部声明。

4. 数据类型、变量、常量

类型概述

本主题提供 Delphi 数据类型的高级概述。

类型本质上是一种数据的名称。声明变量时，必须指定其类型，该类型确定变量可以保存的值集以及可以对其执行的操作。每个表达式都返回特定类型的数据，每个函数也是如此。大多数函数和过程都需要特定类型的参数。

Delphi 语言是一种“强类型”语言，这意味着它区分了各种数据类型，并且并不总是允许您将一种类型替换为另一种类型。这通常是有益的，因为它可以让编译器智能地处理数据并更彻底地验证代码，从而防止难以诊断的运行时错误。但是，当您需要更大的灵活性时，有一些机制可以规避强类型。其中包括类型转换、指针、变体、记录中的变体部分以及变量的绝对寻址。

有几种方法可以对 Delphi 数据类型进行分类：

某些类型是预定义的（或内置的）;编译器会自动识别这些内容，而无需声明。此语言参考中记录的几乎所有类型都是预定义的。其他类型是通过声明创建的;其中包括用户定义的类型和产品库中定义的类型。

类型可分为基本类型和平台类型。基本类型的范围和格式在 Delphi 语言的所有实现中都是相同的，无论底层 CPU 和操作系统如何。平台类型的范围和格式是特定于平台的，并且可能因不同的实现而异。大多数预定义类型是基本类型，但少数整数、字符、字符串和指针类型是平台的。最好尽可能使用平台类型，因为它们可提供最佳性能和可移植性。但是，将存储格式从一个平台类型的实现更改为下一个实现可能会导致兼容性问题 - 例如，如果将内容作为原始二进制数据流式传输到文件，而没有类型和版本控制信息。

类型可以分为简单类型、字符串类型、结构化类型、指针类型、过程类型或变体类型。此外，类型标识符本身可以被视为属于特殊的“类型”，因为它们可以作为参数传递给某些函数（如 High，Low 和 SizeOf）。

类型也可以参数化或泛型。类型可以是泛型的，因为它们是结构或过程的基础，该结构或过程与稍后确定的不同类型协同运行。有关泛型或参数化类型的详细信息，请参阅泛型索引。

下面的大纲显示了 Delphi 数据类型的分类：

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

标准函数 SizeOf 对所有变量和类型标识符进行操作。它返回一个整数，表示存储指定类型的数据所需的内存量（以字节为单位）。例如：

在 32 位平台和 64 位 Windows SizeOf（LongInt） 中返回 4，因为 LongInt 变量使用四个字节的内存。

在 64 位 iOS 中，SizeOf（LongInt） 返回 8，因为 LongInt 变量使用 8 个字节的内存。

以下主题演示了类型声明。有关类型声明的一般信息，请参阅数据类型、变量和常量索引（Delphi）。

简单类型

简单类型（包括序数类型和实数类型）定义有序的值集。

序数类型

序号类型包括整数、字符、布尔值、枚举和子范围类型。序号类型定义一组有序的值，其中除第一个值外的每个值都有一个唯一的前置值，除最后一个值外的每个值都有一个唯一的后继值。此外，每个值都有一个序号，该序号确定类型的顺序。在大多数情况下，如果一个值具有序数 n，则其前置值具有序数 n-1，其后继值具有序数 n+1。

对于整数类型，值的序号是值本身。子范围类型保持其基类型的顺序。对于其他序号类型，默认情况下，第一个值的序数为 0，下一个值的序数为 1，依此类推。枚举类型的声明可以显式重写此默认值。

几个预定义函数对序号值和类型标识符进行操作。其中最重要的总结如下。

函数	参数	返回值	说明
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.

例如，High（Byte） 返回 255，因为 Byte 类型的最大值是 255，而 Succ（2） 返回 3，因为 3 是 2 的后继值。

标准过程 Inc 和 Dec 递增和递减序号变量的值。例如，Inc（I） 等价于 I ： = Succ（I），如果 I 是整数变量，则等价于 I ： = I + 1。

整数类型

整数类型表示整数的子集。

整数类型可以依赖于平台，也可以独立于平台。

依赖平台的整数类型

此整数类型会转换为平台相关的整数类型以适合当前编译器平台的位大小。依赖于平台的整数类型是 NativeInt、NativeUInt、LongInt 和 LongWord。尽可能使用这些类型，因为它们可以为底层 CPU 和操作系统带来最佳性能。下表说明了 Delphi 编译器的范围和存储格式。

类型	平台	值范围	格式	别名
NativeInt	32-bit platforms	-2147483648..2147483647 (-231..2^31-1)	Signed 32-bit	Integer
	64-bit platforms	-9223372036854775808..9223372036854775807 (-263..263-1)	Signed 64-bit	Int64
NativeUInt	32-bit platforms	0..4294967295 (0..232-1)	Unsigned 32-bit	Cardinal
	64-bit platforms	0..18446744073709551615 (0..264-1)	Unsigned 64-bit	UInt64

LongInt	32-bit platforms 64-bit Windows platforms	-2147483648..2147483647 (-231..231-1)	Signed 32-bit	Integer
	64-bit POSIX platforms including iOS, macOS, Android, and Linux	-9223372036854775808..9223372036854775807 (-263..263-1)	Signed 64-bit	Int64
LongWord	32-bit platforms and 64-bit Windows platforms	0..4294967295 (0..232-1)	Unsigned 32-bit	Cardinal
	64-bit POSIX platforms including iOS, macOS, Android, and Linux	0..18446744073709551615 (0..264-1)	Unsigned 64-bit	UInt64

注意：32 位平台包括 32 位 Windows 和 32 位 Android。

独立于平台的整数类型

与平台无关的整数类型始终具有相同的大小，无论您使用什么平台。独立于平台的整数类型包括 ShortInt、SmallInt、LongInt、Integer、Int64、Byte、Word、LongWord、Cardinal 和 UInt64。

类型	值范围	格式	别名
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 (-263..263-1)	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..264-1)	Unsigned 64-bit	

通常，对 Integer 类型的算术运算返回 Integer 类型的值，该值等效于 32 位 LongInt。仅当对一个或多个 Int64 操作数执行时，操作才返回 Int64 类型的值。因此，以下代码会产生不正确的结果：

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

若要在此情况下获取 Int64 返回值，请将 I 转换为 Int64：

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

有关详细信息，请参《阅算术运算符》。

注意：某些采用整数参数的标准例程将 Int64 值截断为 32 位。但是，High、Low、Succ、Pred， Inc、Dec、IntToStr 和 IntToHex 例程完全支持 Int64 参数。此外，Round、Trunc、StrToInt64 和 StrToInt64Def 函数返回 Int64 值。一些例程根本无法采用 Int64 值。

当您递增整数类型的最后一个值或递减第一个值时，结果将环绕范围的开头或结尾。例如，ShortInt 类型的范围为 -128..127;因此，在执行代码后：

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

I 的值为 -128。但是，如果启用了编译器范围检查，则此代码将生成运行时错误。

字符类型

字符类型包括 Char、AnsiChar、WideChar、UCS2Char 和 UCS4Char：

- ✓ 当前实现中的 char 等效于 WideChar，因为现在默认字符串类型是 UnicodeString。由于 Char 的实现可能会在未来版本中发生变化，因此在编写可能需要处理不同大小字符的程序时，最好使用标准函数 SizeOf 而不是硬编码常量。
- ✓ AnsiChar 值是根据区域设置字符集（可能是多字节）排序的字节大小（8 位）字符。
- ✓ WideChar 使用多个字节来表示每个字符。在当前的实现中，WideChar 是根据 Unicode 字符集排序的字大小（16 位）字符（请注意，在将来的实现中可能会更长）。前 256 个 Unicode 字符对应于 ANSI 字符。
- ✓ UCS2Char 是 WideChar 的别名。
- ✓ UCS4Char 用于处理 4 字节 Unicode 字符。

长度为 1 的字符串常量（如“A”）可以表示字符值。预定义函数 Chr 返回 WideChar 范围内任何整数的字符值;例如，Chr（65） 返回字母 A。

AnsiChar 和 WideChar 值（如整数）在递减或递增超过其范围的开始或结束时环绕（除非启用了范围检查）。例如，执行代码后：

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

字母的值为 A （ASCII 65）。

布尔类型

4 种预定义的布尔类型是 Boolean, ByteBool、WordBool 和 LongBool。Boolean 是首选类型。其他存在是为了提供与其他语言和操作系统库的兼容性。

布尔变量占用一个字节的内存，一个 ByteBool 变量也占用一个字节，一个 WordBool 变量占用 2 个字节（一个 Word），一个 LongBool 变量占用 4 个字节（2 个 Word）。

布尔值由预定义的常量 True 和 False 表示。以下关系成立：

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

当 ByteBool、LongBool 或 WordBool 类型的值的序数为非零时，该值被视为 True。如果此类值出现在需要布尔值的上下文中，编译器会自动将非零序号的任何值转换为 True。

前面的注释指的是布尔值的序数，而不是值本身。在 Delphi 中，布尔表达式不能等同于整数或实数。因此，如果 X 是整数变量，则语句：

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

生成编译错误。将变量强制转换为布尔类型是不可靠的，但以下每种替代方法都有效。

```
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 ...;
```

枚举类型

枚举类型通过简单地列出表示这些值的标识符来定义一组有序的值。这些值没有固有的含义。若要声明枚举类型，请使用以下语法：

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

其中 typeName 和每个值都是有效的标识符。例如，声明：

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

定义一个名为 Suit 的枚举类型，其可能的值为 Club, Diamond, Heart, Spade，其中 Ord（Club） 返回 0，Ord（Diamond） 返回 1，依此类推。

声明枚举类型时，将每个值声明为类型为名称的常量。如果此名称标识符用于同一范围内的其他目的，则会发生命名冲突。例如，假设您声明了以下类型：

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

遗憾的是，Click 也是为 TControl 定义的方法的名称，以及 VCL 中从它衍生的所有对象。因此，如果您正在编写应用程序并创建一个事件处理程序，例如：

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

你会得到一个编译错误;编译器将过程范围内的单击解释为对 TForm 的 Click 方法的引用。您可以通过限定标识符来解决此问题;因此，如果在 MyUnit 中声明了 TSound，您将使用：

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

但是，更好的解决方案是选择不太可能与其他标识符冲突的常量名称。例子：

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

您可以直接在变量声明中使用 （val1， ...， valn） 构造，就好像它是类型名称一样：

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

但是，如果以这种方式声明 MyCard，则不能使用这些常量标识符在同一范围内声明另一个变量。因此：

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

生成编译错误。但如下代码：

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

编译正常，和如下代码等同：

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

枚举类型自定义序数值

默认情况下，枚举值的序号从 0 开始，并遵循其标识符在类型声明中列出的顺序。可以通过为声明中的部分或全部值显式分配序号来重写此字段。若要为值赋值序号，请在其标识符后面跟 = constantExpression，其中 constantExpression 是计算结果为整数的常量表达式。例如：

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

定义一个名为 Size 的类型，其可能的值包括“Small ”、“Medium ”和“Large ”，其中 Ord（Small） 返回 5，Ord（Medium） 返回 10，Ord（Large） 返回 15。

枚举类型实际上是一个子范围，其最低值和最大值对应于声明中常量的最低和最高序数。在前面的示例中，Size 类型有 11 个可能的值，其序数范围为 5 到 15。（因此，Char 的类型 array[Size] 表示一个包含 11 个字符的数组。其中只有三个值具有名称，但其他值可通过类型转换和例程（如 Pred、Succ、Inc 和 Dec）访问。在下面的示例中，Size 范围内的“匿名”值分配给变量 X。

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

任何未显式分配序数的值的序数都大于列表中前一个值的序数。如果未为第一个值分配序数，则其序数为 0。因此，鉴于声明：

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

SomeEnum 只有两个可能的值：Ord（e1） 返回 0，Ord（e2） 返回 1，Ord（e3） 也返回 1;由于 E2 和 E3 具有相同的序数，因此它们表示相同的值。

没有特定值的枚举常量具有 RTTI：

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

而具有特定值的枚举常量（如下所示）没有 RTTI：

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

作用域枚举

如果启用 {$SCOPEDENUMS ON} 编译器指令，则可以在 Delphi 代码中使用作用域枚举。

{$SCOPEDENUMS ON 或 OFF} 编译器指令启用或禁用在 Delphi 代码中使用作用域枚举。{$SCOPEDENUMS ON} 定义枚举的作用域。{$SCOPEDENUMS ON} 会影响枚举类型的声明，直到最近的 {$SCOPEDENUMS OFF} 指令。在 {$SCOPEDENUMS ON} 指令之后声明的枚举类型中引入的枚举标识符不会添加到全局范围。若要使用作用域枚举标识符，应使用引入此标识符的枚举类型的名称对其进行限定。

例如，让我们在 Unit1.pas 文件中定义以下枚举：

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

以及以下使用此单元的程序：

```
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.
```

现在，我们可以研究 {$SCOPEDENUMS} 编译器指令对 TMyEnum 枚举中定义的 First、Second 和 Third 标识符可见的作用域的影响。

首先，在此代码上运行 （F9）。代码成功运行。这意味着 First 标识符，用于

```
Value := First;
```

此变量是全局范围内标识符，在此引入：
TMyEnum = (First, Second, Third);

的枚举类型。
现在，清除以下注释//

{$SCOPEDENUMS ON}

Unit1 单元中的编译器指令。此指令强制限定 TMyEnum 枚举的作用域。执行运行 The E2003 Undeclared identifier 'First'错误在：

Value := First;

行，它通知 {$SCOPEDENUMS ON} 编译器指令阻止将作用域 TMyEnum 枚举中引入的 First 标识符添加到全局范围。
若要使用作用域枚举中引入的标识符，请在对枚举元素的引用前面加上其类型名称。例如，在第二个中清除注释

Value := TMyEnum.First;

值变量的版本（并注释值的第一个版本）。执行运行。程序运行成功。这意味着 First 标识符在 TMyEnum 范围内是已知的。
现在添加如下注释//

// {$SCOPEDENUMS ON}

Unit1 中的编译器指令。然后从第一个变量的声明中清除注释

First: Integer;

并再次使用

Value := First;

变量，现在，程序 Project1 中的代码如下所示：

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

执行运行：

```
First: Integer;
```

行导致 E2010 Incompatible types - 'TMyEnum' and 'Integer' error 错误。这意味着命名冲突发生在 TMyEnum 枚举中引入的全局范围第一个标识符和第一个变量之间。可以通过将第一个标识符与定义它的 unit1 单元限定来解决此问题。为此，再次注释 Value 变量的第一个版本，并清除第三个版本的注释：

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

执行运行。程序运行成功。也就是说，现在可以使用 unit1 单元范围限定 First 标识符。但是，如果我们再次启用

```
{$SCOPEDENUMS ON}
```

unit1 中的编译器指令。编译器将产生 E2003 Undeclared identifier 'First' 错误，在

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

行，这意味着 {$SCOPEDENUMS ON} 阻止在 unit1 作用域中添加 First 枚举的标识符。现在，First 标识符仅添加到 TMyEnum 枚举的范围内。要检查这一点，让我们再次使用：

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

值变量的版本。执行“运行”，代码成功。

子范围类型（子界类型）

子范围类型表示另一个序号类型（称为基类型）中的值的子集。任何形式的结构 Low..High，其中 Low 和 High 是同一序号类型的常量表达式，Low 小于 High，标识包含低和高之间的所有值的子范围类型。例如，如果声明枚举类型：

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

然后，您可以定义子范围类型，如下所示：

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

此处的 TMyColors 包括 Green, Yellow, Orange, Purple, White 的值。

您可以使用数字常量和字符（长度为 1 的字符串常量）来定义子范围类型：

```
type
 SomeNumbers = -128..127;
```

```
Caps = 'A'..'Z';
```

使用数字或字符常量定义子范围时，基类型是包含指定范围的最小整数或字符类型。

LowerBound..UpperBound 构造本身充当类型名称，因此可以直接在变量声明中使用它。例如：

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

声明一个整数变量，其值可以是 1 到 500 之间的任意值。

从基类型中保留子范围中每个值的序号。（在第一个示例中，如果 Color 是保存值 Green 的变量，则无论 Color 的类型是 TColors 还是 TMyColors，Ord（Color） 都返回 2。值不会环绕子范围的开头或结尾，即使基数是整数或字符类型也是如此;递增或递减超过子范围的边界只是将值转换为基类型。因此，虽然：

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

将生成错误，如下代码：

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

将值 100 分配给 I（除非启用了编译器范围检查）。

在子范围定义中使用常量表达式引入了语法困难。在任何类型声明中，当 = 之后的第一个有意义的字符是左括号时，编译器假定正在定义枚举类型。因此代码：

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

将产生错误。通过重写类型声明来解决此问题，以避免使用前导括号：

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

实数类型

实数类型定义一组可以用浮点表示法表示的数字。下表给出了 64 位和 32 位平台上实际类型的范围和存储格式。

类型	平台	范围（正数）	十进制有效位	占用字节
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	32-bit Intel Windows	3.37e-4932 .. 1.18e+4932	10-20	10
	64-bit Intel Linux 64-bit Intel macOS	3.37e-4932 .. 1.18e+4932	10-20	16
	64-bit Intel Windows ARM platforms (32-bit Android, 64-bit Android, 64-bit iOS, 64-bit macOS)	2.23e-308 .. 1.79e+308	15-16	8
Comp	all	-9223372036854775808.. 9223372036854775807 (-263.. 263-1)	10-20	8
Currency	all	-922337203685477.5808.. 922337203685477.5807 (-(263+1)/10000.. 263/10000)	10-20	8

以下条目适用于实数类型：

- ✓ 在当前的实现中，Real 等同于 Double。
- ✓ 维护 Real48 是为了向后兼容。由于它的存储格式不是英特尔处理器架构的原生格式，因此它会导致性能比其他浮点类型慢。6 字节 Real48 类型在 Object Pascal 的早期版本中被称为 Real。如果要重新编译在 Delphi 中使用较旧的 6 字节 Real 类型的代码，则可能需要将其更改为 Real48。还可以使用 {$REALCOMPATIBILITY ON} 编译器指令将 Real 转换回 6 字节类型。
- ✓ Extended 在 32 位平台上提供比其他实际类型更高的精度。在 64 位 Windows 和所有 ARM 平台上，扩展是 Double 的别名;也就是说，扩展数据类型的大小为 8 个字节。因此，与 32 位平台（其中扩展为 10 个字节）相比，在这些平台上使用扩展的精度较低。因此，如果应用程序使用扩展数据类型，并且依赖于浮点运算的精度，则此大小差异可能会影响数据。如果要创建要跨平台共享的数据文件，请谨慎使用 Extended。有关详细信息，请参阅 64 位 Windows 系统上的 Extended 数据类型小 2 个字节。
- ✓ Comp（计算）类型是英特尔处理器架构的原生类型，表示 64 位整数。但是，它被归类为实数，因为它的行为不像序数类型。（例如，不能递增或递减“补偿”值。维护 Comp 只是为了向后兼容。使用 Int64 类型以获得更好的性能。
- ✓ Currency 是一种定点数据类型，可最大限度地减少货币计算中的舍入误差。它存储为缩放的 64 位整数，其中 4 个最低有效数字隐式表示小数位。当在赋值和表达式中与其他实数类型混合时，货币值会自动除以或乘以 10000。

字符串

本主题介绍 Delphi 语言中可用的字符串数据类型。涵盖以下类型：

短字符串（ShortString）

ANSI 字符串 （AnsiString）
Unicode 字符串（UnicodeString 和 WideString）

字符串概述

字符串表示字符序列。Delphi 支持以下预定义的字符串类型。

类型	最大长度	内存占用	使用场景
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.

注意：默认字符串类型为 UnicodeString。提供 WideString 是为了与 COM BSTR 类型兼容。通常应将 UnicodeString 用于非 COM 应用程序。在大多数情况下，UnicodeString 是首选类型。string 类型是 UnicodeString 的别名。

字符串类型可以在赋值和表达式中混合;编译器会自动执行所需的转换。但是通过引用传递给函数或过程（作为 var 和 out 参数）的字符串必须是适当的类型。字符串可以显式转换为不同的字符串类型。但是，将多字节字符串转换为单字节字符串可能会导致数据丢失。

有一些特殊的字符串类型值得一提：

- ✓ 代码分页的 AnsiStrings 定义如下：
  Type mystring = type AnsiString(CODEPAGE)
  它是一个 AnsiString，与在特定代码页中维护其内部数据具有关联性。
- ✓ RawByteString 类型为 AnsiString（$FFFF） 类型。RawByteString 允许传递任何代码页的字符串数据，而无需执行任何代码页转换。RawByteString 只能用作 const 或值类型参数或函数的返回类型。它永远不应该通过引用传递（通过 var 传递），也不应该被实例化为变量。
- ✓ UTF8String 表示使用 UTF-8（可变字节数 Unicode）编码的字符串。它是具有 UTF-8 代码页的代码分页 AnsiString 类型。

保留字字符串的功能类似于常规字符串类型标识符。例如：

```
var S: string;
```

创建一个保存字符串的变量 S。在 Win32 平台上，编译器将字符串（当它后面没有括号的数字出现时）解释为 UnicodeString。

在 Win32 平台上，可以使用 {$H-} 指令将字符串转换为 ShortString 。在当前程序中使用较旧的 16 位 Delphi 代码或 Turbo Pascal 代码时，这是一种可能有用的技术。

请注意，在声明特定长度的 ShortString 类型时，也会使用关键字 string（请参阅下面的 ShortString ）。

字符串的比较由相应位置的元素顺序定义。在不等长的字符串之间，较长字符串中没有较短字符串中相应字符的每个字符都具有大于值。例如，“AB”大于“A”;也就是说，“AB”>“A”返回 True。零长度字符串表示最小值。

您可以像索引数组一样为字符串变量编制索引。如果 S 是非 UnicodeString 字符串变量，而 i 是整数表达式，则 S[i] 表示 S 中的第 i 个字节，对于多字节字符串 （MBCS），它可能不是第 i 个字符或整个字符。同样，为 UnicodeString 变量编制索引会导致元素不是整个字符。如果字符串包含基本多语言平面 （BMP） 中的字符，则所有字符均为 2 个字节，因此为字符串编制索引会获取字符。但是，如果某些字符不在 BMP 中，则索引元素可能是代理项对 - 而不是整个字符。

标准函数 Length 返回字符串中的元素数。如上所述，元素的数量不一定是字符的数量。SetLength 过程调整字符串的长度。请注意，SizeOf 函数返回用于表示变量或类型的字节数。请注意，SizeOf 仅返回短字符串的字符串中的字符数。SizeOf 返回所有其他字符串类型的指针中的字节数，因为它们是指针。

对于短字符串或 AnsiString，S[i] 的类型为 AnsiChar。对于 WideString，S[i] 的类型是 WideChar。对于单字节（西方）语言环境，MyString[2] ：= 'A';将值 A 分配给 MyString 的第二个字符。以下代码使用标准 UpCase 函数将 MyString 转换为大写：

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

请小心以这种方式为字符串编制索引，因为覆盖字符串的末尾可能会导致访问冲突。此外，请避免将字符串索引作为 var 参数传递，因为这会导致代码效率低下。

可以将字符串常量（或返回字符串的任何其他表达式）的值分配给变量。进行赋值时，字符串的长度会动态更改。例子：

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

```
MyString := '';  { empty string }
```

ShortString

短字符串的长度为 0 到 255 个单字节字符。虽然 ShortString 的长度可以动态变化，但其内存是静态分配的 256 字节;第一个字节存储字符串的长度，其余 255 个字节可用于字符。如果 S 是 ShortString 变量，则 Ord（S[0]） 与 Length（S） 一样，返回 S 的长度;将值分配给 S[0]，就像调用 SetLength 一样，会更改 S 的长度。保留 ShortString 只是为了向后兼容。

Delphi 语言支持短字符串类型 - 实际上是 ShortString 的子类型 - 其最大长度介于 0 到 255 个字符之间。这些由附加到保留字字符串的括号数字表示。例如：

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

创建一个名为 MyString 的变量，其最大长度为 100 个字符。这等效于声明：

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

以这种方式声明的变量仅分配类型所需的内存量 - 即指定的最大长度加上一个字节。在我们的示例中，MyString 使用 101 个字节，而预定义 ShortString 类型的变量使用 256 个字节。

为短字符串变量赋值时，如果字符串超过类型的最大长度，则该字符串将被截断。

标准函数 High 和 Low 对短字符串类型标识符和变量进行操作。High 返回短字符串类型的最大长度，而 Low 返回零。

AnsiString

AnsiString 表示动态分配的字符串，其最大长度仅受可用内存的限制。

AnsiString 变量是包含字符串信息的结构。当变量为空时 - 即，当它包含零长度字符串时，指针为 nil，并且字符串不使用额外的存储空间。当变量为非空时，它指向包含字符串值的动态分配的内存块。此内存在堆上分配，但其管理是完全自动的，不需要用户代码。AnsiString 结构包含一个 32 位长度指示符、一个 32 位引用计数、一个指示每个字符字节数的 16 位数据长度和一个 16 位代码页。

AnsiString 表示单个字节字符串。对于单字节字符集 （SBCS），字符串中的每个字节表示一个字符。在多字节字符集 （MBCS） 中，元素仍然是单个字节，但某些字符由一个字节表示，而其他字符由多个字节表示。多字节字符集 - 特别是双字节字符集 （DBCS） - 广泛用于亚洲语言。AnsiString 可以包含 MBCS 字符。

AnsiString 的索引是从 1 开始的。索引多字节字符串是不可靠的，因为 S[i] 表示 S 中的第 i 个字节（不一定是第 i 个字符）。第 i 个字节可以是单个字符或字符的一部分。但是，标准的 AnsiString 字符串处理函数具有启用多字节的对应项，这些对应函数还实现了特定于区域设置的字符排序。（多字节函数的名称通常以 Ansi- 开头。例如，StrPos 的多字节版本是 AnsiStrPos。多字节字符支持取决于操作系统，并基于当前区域设置。

由于 AnsiString 变量具有指针，因此其中两个或多个变量可以引用相同的值，而不会消耗

额外的内存。编译器利用这一点来节省资源并更快地执行分配。每当销毁 AnsiString 变量或为其分配新值时，旧 AnsiString 的引用计数（变量的先前值）都会递减，新值（如果有）的引用计数会递增;如果字符串的引用计数达到零，则解除分配其内存。此过程称为引用计数。当使用索引来更改字符串中单个字符的值时，如果（但仅当）字符串的引用计数大于 1，则会创建字符串的副本。这称为写入时复制语义。

UnicodeString

UnicodeString 类型是默认字符串类型，表示动态分配的 Unicode 字符串，其最大长度仅受可用内存的限制。

在 Unicode 字符集中，每个字符由一个或多个字节表示。Unicode 有几种 Unicode 转换格式，这些格式使用不同但等效的字符编码，可以很容易地相互转换。

- ✓ 例如，在 UTF-8 中，字符可以是 1 到 4 个字节。在 UTF-8 中，前 128 个 Unicode 字符映射到 US-ASCII 字符。
- ✓ UTF-16 是另一种常用的 Unicode 编码，其中字符为 2 个字节或 4 个字节。世界上大多数字符都在基本多语言平面中，可以用 2 个字节表示。其余字符需要两个称为代理项对的 2 字节字符。
- ✓ UTF-32 表示每个字符，每个字符有 4 个字节。
- ✓

Win32 平台支持单字节和多字节字符集以及 Unicode。Windows 操作系统支持 UTF-16。有关详细信息，请参阅 Unicode 标准。

UnicodeString 类型与 AnsiString 类型具有完全相同的结构。UnicodeString 数据以 UTF-16 编码。

由于 UnicodeString 和 AnsiString 具有相同的结构，因此它们的功能非常相似。当 UnicodeString 变量为空时，它不会使用额外的内存。当它不为空时，它指向包含字符串值的动态分配的内存块，并且对此的内存处理对用户是透明的。UnicodeString 变量是引用计数的，其中两个或多个变量可以引用相同的值，而不会消耗额外的内存。

UnicodeString 的实例可以索引字符。索引是从 1 开始的，就像 AnsiString 一样。

UnicodeString 与所有其他字符串类型兼容赋值。但是，AnsiString 和 UnicodeString 之间的赋值会执行适当的向上或向下转换。请注意，不建议将 UnicodeString 类型分配给 AnsiString 类型，这可能会导致数据丢失。

Delphi 还可以通过 WideChar，PWideChar 和 WideString 类型支持 Unicode 字符和字符串。

有关使用 Unicode 的更多信息，请参阅 RAD Studio 中的 Unicode 和为 Unicode 启用应用程序。

WideString

WideString 类型表示动态分配的 16 位 Unicode 字符字符串。在某些方面，它类似于 AnsiString。在 Win32 上，WideString 与 COM BSTR 类型兼容。

宽字符串适用于 COM 应用程序。但是，WideString 不被引用计数，因此 UnicodeString 在其他类型的应用程序中更加灵活和高效。

宽字符串多字节字符串的索引是不可靠的，因为 S[i] 表示 S 中的第 i 个元素（不一定是

第 i 个字符）。

对于 Delphi，Char 和 PChar 类型分别是 WideChar 和 PWideChar 类型。

注意：用于移动平台的 Delphi 编译器不支持 WideString，但用于桌面平台的 Delphi 编译器使用 WideString。

处理 null 结尾的字符串

许多编程语言（包括 C 和 C++）都缺少专用的字符串数据类型。这些语言以及使用它们构建的环境依赖于以 null 结尾的字符串。以 null 结尾的字符串是从零开始的字符数组，以 NUL 结尾;由于数组没有长度指示符，因此第一个 NUL 字符标记字符串的结尾。当您需要与使用它们的系统共享数据时，您可以使用 SysUtils 单元中的 Delphi 构造和特殊例程（请参阅标准例程和输入输出）来处理以 null 结尾的字符串。

例如，以下类型声明可用于存储以 null 结尾的字符串：

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

启用扩展语法 （{$X+}） 后，可以将字符串常量分配给静态分配的从零开始的字符数组。（动态数组不适用于此目的。）如果使用短于数组声明长度的字符串初始化数组常量，则其余字符将设置为 #0。

字符指针、数组、字符串常量

若要操作以 null 结尾的字符串，通常需要使用指针。（请参阅指针和指针类型 （Delphi））。字符串常量与 PChar 和 PWideChar 类型赋值兼容，这些类型表示指向以 null 结尾的 Char 和 WideChar 值数组的指针。例如：

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

将 P 指向包含原始常量字符串“Hello world！这相当于：

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

您还可以将字符串常量传递给任何采用 PChar 或 PWideChar 类型的值或常量参数的函数 - 例如 StrUpper（'Hello world！'）。与对 PChar 的赋值一样，编译器生成字符串的 null 终

止副本，并为函数提供指向该副本的指针。最后，您可以使用字符串文本单独或以结构化类型初始化 PChar 或 PWideChar 常量。例子：

```
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');
```

从零开始的字符数组与 PChar 和 PWideChar 兼容。使用字符数组代替指针值时，编译器会将数组转换为指针常量，其值对应于数组中第一个元素的地址。例如：

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

此代码使用相同的值调用 SomeProcedure 两次。

字符指针可以像数组一样编制索引。在前面的示例中，MyPointer[0] 返回 H。索引指定在取消引用指针之前添加到指针的偏移量。（对于 PWideChar 变量，索引会自动乘以 2。因此，如果 P 是字符指针，P[0] 等效于 P^ 并指定数组中的第一个字符，P[1] 指定数组中的第二个字符，依此类推;P[-1] 指定紧邻 P[0] 左侧的“字符”。编译器不对这些索引执行范围检查。

StrUpper 函数演示了如何使用指针索引来循环访问以 null 结尾的字符串：

```
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;
```

混合使用 String 和 null 结尾字符串

可以在表达式和赋值中混合字符串（AnsiString 和 UnicodeString 值）和以 null 结尾的字符串（PChar 值），并且可以将 PChar 值传递给采用字符串参数的函数或过程。赋值 S ：= P，其中 S 是字符串变量，P 是 PChar 表达式，它将以 null 结尾的字符串复制到字符串中。

在二元操作中，如果一个操作数是字符串，另一个是 PChar，则 PChar 操作数将转换为 UnicodeString。

您可以将 PChar 值转换为 UnicodeString。当您要对两个 PChar 值执行字符串操作时，这很有用。例如：

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

还可以将 UnicodeString 或 AnsiString 字符串转换为以 null 结尾的字符串。以下规则适用：

- ✓ 如果 S 是 UnicodeString，PChar（S） 将 S 转换为以 null 结尾的字符串;它返回指向 S 中第一个字符的指针。此类强制转换用于 Windows API。例如，如果 Str1 和 Str2 是 UnicodeString，则可以像这样调用 Win32 API MessageBox 函数：
  MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
  如果 S 是 AnsiString，请使用 PAnsiChar（S）。
- ✓ 还可以使用指针将字符串强制转换为非类型化指针。但如果 S 为空，则类型转换返回 nil。
- ✓ PChar（S） 始终返回指向内存块的指针;如果 S 为空，则返回指向 #0 的指针。
- ✓ 将 UnicodeString 或 AnsiString 变量强制转换为指针时，指针将保持有效，直到为变量分配新值或超出范围。如果将任何其他字符串表达式强制转换为指针，则指针仅在执行类型转换的语句中有效。
- ✓ 将 UnicodeString 或 AnsiString 表达式强制转换为指针时，指针通常应被视为只读。仅当满足以下所有条件时，才能安全地使用指针修改字符串：

  - ✓ 表达式强制转换是 UnicodeString 或 AnsiString 变量。
  - ✓ 字符串不为空。
  - ✓ 字符串是唯一的 - 即具有 1 的引用计数。若要保证字符串是唯一的，请调用 SetLength、SetString 或 UniqueString 过程。
  - ✓ 自创建类型转换以来，字符串未被修改。
  - ✓ 修改的字符都在字符串中。注意不要在指针上使用超出范围的索引。

将 WideString 值与 PWideChar 值混合时，相同的规则适用。

复合结构类型

结构化类型的实例包含多个值。结构化类型包括集合、数组、记录和文件以及类、类引用和接口类型。除了仅保存序号值的集合外，结构化类型可以包含其他结构化类型;一个类型可以具有无限级别的结构。

本主题介绍以下结构化类型：

集合
数组，包括静态和动态数组
记录
文件类型

内存对齐

默认情况下，结构化类型中的值在单字或双字边界上对齐，以便更快地访问。

但是，可以通过在声明结构化类型时包含 packed 保留字来指定按字节对齐方式。packed 指定压缩数据存储。下面是一个示例声明：

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

不建议使用 packed，因为它会阻止与其他语言或平台的兼容性，减慢数据访问速度，并且在字符数组的情况下，会影响类型兼容性。有关更多信息，请参见内存管理和具有通用类型规范的字段隐式打包。

集合

集合是相同序号类型的值的集合。这些值没有固有的顺序，一个值在一个集合中包含两次也没有意义。

集合类型的范围是特定序数类型的幂集，称为基类型;也就是说，集合类型的可能值是基类型的所有子集，包括空集。基类型不能超过 256 个可能的值，并且它们的序数必须介于 0 和 255 之间。集合的任何构造：

```
set of baseType
```

其中 baseType 是适当的序号类型，标识集合类型。
由于基类型的大小限制，集合类型通常使用子范围定义。例如，声明：

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

创建一个名为 TIntSet 的集合类型，其值是介于 1 到 250 之间的整数集合。您可以使用以下方法完成相同的操作：

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

给定此声明，您可以创建如下集合：

```
var Set1, Set2: TIntSet;
      ...
```

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

您还可以使用 set of ... 直接在变量声明中构造：

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

注意： 有关详细信息，请参阅以下警告消息：W1050 WideChar reduced to byte char in set expressions (Delphi).

集合类型的其他示例包括：

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

in 运算符测试成员身份：

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

每个集合类型都可以保存空集合，用 [] 表示。有关集合的详细信息，请参阅表达式（Delphi） 中的“集合构造函数”和“集合运算符”。

数组

数组表示相同类型（称为基类型）的元素的索引集合。由于每个元素都有一个唯一的索引，因此数组与集合不同，可以多次有意义地包含相同的值。数组可以静态或动态分配。

静态数组

静态数组类型由以下形式的构造表示：

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

其中每个索引类型都是范围不超过 2GB 的序号类型。由于 indexTypes 索引数组，因此数组可以容纳的元素数受 indexType 大小的乘积的限制。实际上，indexType 通常是整数子范围。

在一维数组的最简单情况下，只有一个 indexType。例如：

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

声明一个名为 MyArray 的变量，该变量包含 100 个字符值的数组。鉴于此声明，

MyArray[3] 表示 MyArray 中的第三个字符。如果您创建了一个静态数组，但没有为其所有元素赋值，则未使用的元素仍会被分配并包含随机数据;它们就像未初始化的变量。

多维数组是数组的数组。例如：

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

相当于：

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

无论以何种方式声明 TMatrix，它都表示一个包含 500 个实数值的数组。TMatrix 类型的变量 MyMatrix 可以像这样编制索引：MyMatrix[2,45];或者像这样：MyMatrix[2][45]。同样地：

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

相当于：

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

标准函数 Low 和 High 对数组类型标识符和变量进行操作。它们返回数组第一个索引类型的下限和上限。标准函数 Length 返回数组第一维中的元素数。

Char 值的一维 packed 静态数组称为打包字符串。打包字符串类型与字符串类型以及具有相同元素数的其他打包字符串类型兼容。请参阅《类型兼容和等同 （Delphi）》。

array[0..x] of Char 的数组类型称为从零开始的字符数组。从零开始的字符数组用于存储以 null 结尾的字符串，并且与 PChar 值兼容。请参阅字符串类型 （Delphi） 中的“《使用以 null 结尾的字符串》”。

动态数组

动态数组没有固定的大小或长度。相反，当您为数组分配值或将其传递给 SetLength 过程时，将重新分配动态数组的内存。动态数组类型由以下形式的构造表示：

```
array of baseType
```

例如：

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

声明实数的一维动态数组。该声明不会为 MyFlexibleArray 分配内存。若要在内存中创建数组，请调用 SetLength。例如，给定前面的声明：

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

分配一个包含 20 个实数的数组，索引为 0 到 19。为动态数组分配内存的另一种方法是调用数组构造函数：

```
type
  TMyFlexibleArray = array of Integer;

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

它为三个元素分配内存并为每个元素分配给定的值。
与数组构造函数类似，动态数组也可以从数组常量表达式初始化，如下所示。

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

请注意，与数组构造函数不同，数组常量可以直接应用于未命名的动态数组类型。此语法特定于动态数组;将此技术应用于其他数组类型可能会导致常量被解释为集合，从而导致编译时出现不兼容的类型错误。

动态数组始终是整数索引的，始终从 0 开始。

动态数组变量是隐式指针，由用于长字符串的相同引用计数技术进行管理。要释放动态数组，请将 nil 分配给引用该数组的变量或将变量传递给 Finalize;这些方法中的任何一个都会释放数组，前提是没有对它的其他引用。当动态数组的引用计数降至零时，它们会自动释放。长度为 0 的动态数组的值为 nil。不要将取消引用运算符 （^） 应用于动态数组变量或将其传递给“新建”或“释放”过程。

如果 X 和 Y 是同一动态数组类型的变量，则 X ：= Y 将 X 指向与 Y 相同的数组（在执行此操作之前无需为 X 分配内存。与字符串和静态数组不同，动态数组不采用写入时复制，因此在写入之前不会自动复制它们。例如，执行此代码后：

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

A[0] 的值为 2。（如果 A 和 B 是静态数组，则 A[0] 仍将为 1。

分配给动态数组索引（例如，MyFlexibleArray[2] ：= 7）不会重新分配数组。编译时不会

报告超出范围的索引。

相反，要创建动态数组的独立副本，必须使用全局 Copy 函数：

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

比较动态数组变量时，将比较它们的引用，而不是它们的数组值。因此，在执行代码后：

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

A = B 返回 False，但 A[0] = B[0] 返回 True。

若要截断动态数组，请将其传递给 SetLength，或将其传递给 Copy 并将结果分配回数组变量。（SetLength 过程通常更快。例如，如果 A 是动态数组，则以下任一元素都会截断除 A 的前 20 个元素之外的所有元素：

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

分配动态数组后，可以将其传递给标准函数 Length、High 和 Low。Length 返回数组中的元素数，High 返回数组的最高索引（即 Length - 1），Low 返回 0。在零长度数组的情况下，High 返回 -1（异常结果是 High < Low）。

注意：在某些函数和过程声明中，数组参数表示为 array of baseType，未指定任何索引类型。例如，function CheckStrings(A: array of string): Boolean;

这表示该函数对指定基类型的所有数组进行操作，无论它们的大小、索引方式或它们是静态分配还是动态分配。

多维动态数组

要声明多维动态数组，请迭代使用 array of ... 来构建。例如：

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

声明字符串的二维数组。若要实例化此数组，请使用两个整数参数调用 SetLength。例如，如果 I 和 J 是整数值变量：

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

分配一个 I-by-J 数组，Msgs[0,0] 表示该数组的一个元素。

您可以创建非矩形的多维动态数组。第一步是调用 SetLength，为其传递数组前 n 维的参数。例如：

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

为 Int 分配十行，但不分配列。稍后，您可以一次分配一列（为它们提供不同的长度）；例如：

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

使 Ints 的第三列长度为五个整数。此时（即使尚未分配其他列），您也可以为第三列赋值 - 例如，Ints[2,4] ：= 6。

下面的示例使用动态数组（以及在 SysUtils 单元中声明的 IntToStr 函数）来创建字符串的三角矩阵。

```
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;
```

数组类型和赋值

仅当数组属于同一类型时，它们才与赋值兼容。由于 Delphi 语言对类型使用名称等效性，因此不会编译以下代码。

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

要使赋值正常工作，请将变量声明为：

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

或：

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

动态数组支持的类似字符串的操作

动态数组的操作方式与字符串类似。例如：

```
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;
```

类似字符串的支持例程

除了对字符串的操作外，一些 Delphi 内部例程还支持对动态数组的操作。

System.Insert

函数在位置索引的开头插入一个动态数组。它返回修改后的数组：

```
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

Delete 函数从动态数组中删除元素并返回修改后的数组：

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

System.Concat

Concat 函数可用于将两个不同的动态数组组合在一起：

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

记录（基础）

记录（类似于某些语言中的结构）表示一组异构元素。每个元素称为一个字段;记录类型的声明为每个字段指定名称和类型。记录类型声明的语法为：

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

其中 recordTypeName 是有效的标识符，每种类型表示一种类型，每个字段列表是有效的标识符或逗号分隔的标识符列表。最后一个分号是可选的。

例如，以下声明创建一个名为 TDateRec 的记录类型。

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

每个 TDateRec 包含三个字段：一个名为“Year”的整数值、一个名为“Month”的枚举类型的值，以及另一个介于 1 和 31 之间的整数（称为“Day”）。标识符“Year”、“Month”和“Day”是 TDateRec 的字段指示符，它们的行为类似于变量。但是，TDateRec 类型声明不会为“Year”、“Month”和“Day”字段分配任何内存;实例化记录时会分配内存，如下所示：

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

此变量声明创建 TDateRec 的两个实例，称为 Record1 和 Record2。
您可以通过使用记录名称限定字段指示符来访问记录的字段：

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

或者使用 with 语句：

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

现在，您可以将记录 1 的字段值复制到记录 2：

```
Record2 := Record1;
```

由于字段指示符的范围仅限于其出现的记录，因此您不必担心字段指示符与其他变量之间的命名冲突。
您可以使用 record ... 直接在变量声明中构造：

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

```
end;
```

但是，像这样的声明在很大程度上违背了记录的目的，即避免对类似的变量组进行重复编码。此外，单独申报的这类记录即使结构相同，也不符合赋值兼容。

记录中的变体部分

记录类型可以具有变体部分，该部分类似于 case 语句。变型部件必须遵循记录声明中的其他字段。

若要声明带有变体部件的记录类型，请使用以下语法：

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

声明的第一部分（保留字大小写）与标准记录类型的第一部分相同。声明的其余部分 - 从 case 到可选的最后一个分号 - 称为变体部分。在变体部分：

- ✓ tag 是可选的，可以是任何有效的标识符。如果省略标签，则省略后面冒号(:)。
- ✓ ordinalType 表示序号类型。
- ✓ 每个常量列表都是一个常量，表示 ordinalType 类型的值，或此类常量的逗号分隔列表。任何值都不能在组合的常量列表中多次表示。
- ✓ 每个变体都是一个以分号分隔的声明列表，类似于 fieldList：记录类型主要部分中的类型构造。也就是说，变体的形式为：

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

其中每个字段列表是有效的标识符或逗号分隔的标识符列表，每种类型表示一种类型，最后一个分号是可选的。类型不得是长字符串、动态数组、变体（即变体类型）或接口，也不能是包含长字符串、动态数组、变体或接口的结构化类型;但它们可以是指向这些类型的指针。

具有变体部分的记录在语法上很复杂，但在语义上看似简单。记录的变体部分包含多个变体，这些变体在内存中共享相同的空间。您可以随时读取或写入任何变体的任何字段;但是，如果您先写入一个变体中的字段，然后写入另一个变体中的字段，则可能会覆盖自己的数据。标记（如果有）在记录的非变量部分中用作额外的字段（类型为 ordinalType）。

变型部分有两个用途。首先，假设您要创建一个记录类型，该记录类型具有用于不同类型

数据的字段，但您知道永远不需要在单个记录实例中使用所有字段。例如：

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

这里的想法是，每个员工都有年度工资或小时工资，但不能两者兼而有之。因此，当您创建 TEmployee 实例时，没有理由为这两个字段分配足够的内存。在这种情况下，变体之间的唯一区别在于字段名称，但字段可能很容易具有不同的类型。考虑一些更复杂的示例：

```
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;
```

对于每个记录实例，编译器分配足够的内存来保存最大变体中的所有字段。可选标记和 constantLists（如上一个示例中的矩形、三角形等）在编译器管理字段的方式中没有任何作用；它们的存在只是为了方便程序员。

变体部分的第二个原因是，它们允许您将相同的数据视为属于不同类型的数据，即使在编译器不允许类型转换的情况下也是如此。例如，如果您将 64 位 Real 作为一个变体中的第一个字段，将 32 位整数作为另一个变体中的第一个字段，则可以将一个值分配给 Real 字段，然后将其前 32 位作为 Integer 字段的值读回（例如，传递它， 到需要整数参数的函数）。

记录（高级）

除了传统的记录类型之外，Delphi 语言还允许更复杂和“类状”的记录类型。除了字段之外，记录还可能具有属性和方法（包括构造函数）、类属性、类方法、类字段和嵌套类型。有关这些主题的详细信息，请参阅有关类和对象 （Delphi） 的文档。下面是具有一些“类”功能的示例记录类型定义。

```
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;
```

尽管记录现在可以共享类的大部分功能，但类和记录之间存在一些重要差异。

- ✓ 记录不支持继承。
- ✓ 记录可以包含变体部分;类不能。
- ✓ 记录是值类型，因此它们在赋值时复制、按值传递并在堆栈上分配，除非使用“New”和“Dispose”函数全局声明或显式分配它们。类是引用类型，因此它们不会在赋值时复制，而是通过引用传递，并且在堆上分配。
- ✓ 记录允许运算符重载;但是，类不允许运算符重载。
- ✓ 记录是使用默认的无参数构造函数自动构造的，但必须显式构造类。由于记录具有默认的无参数构造函数，因此任何用户定义的记录构造函数都必须具有一个或多个参数。
- ✓ 记录类型不能有析构函数。
- ✓ 虚拟方法（virtual, dynamic, and message 指定的方法）不能在记录类型中使用。

✓ 与类不同，记录类型平台无法实现接口。

有关记录的更多信息，请参阅有关《自定义托管记录》的文档。

自定义托管记录

Delphi 中的记录可以包含任何数据类型的字段。当记录具有普通（非托管）字段（如数值或其他枚举值）时，编译器无需执行太多操作。创建和处置记录包括分配内存或删除内存位置。

如果记录具有由编译器管理的类型的字段（如字符串或接口），则编译器需要注入额外的代码来管理初始化或完成。例如，字符串是引用计数的，因此当记录超出范围时，记录中的字符串需要减少其引用计数，这可能会导致取消分配字符串的内存。因此，当您在代码的某个部分中使用此类托管记录时，编译器会自动在该代码周围添加一个 try-finally 块，并确保即使在出现异常的情况下也清除数据。这种情况已经持续了很长时间。换句话说，托管记录一直是 Delphi 语言的一部分。

初始化和销毁运算符

具有初始化和销毁运算符的记录类型支持自定义初始化和销毁，超出了编译器对托管记录执行的默认操作。您可以使用自定义初始化和销毁代码声明记录，而不管其字段的数据类型如何，并且可以编写此类自定义初始化和销毁代码。这是通过向记录类型添加特定的新运算符来实现的（如果需要，可以有一个运算符而没有另一个运算符）。下面是一个简单的代码片段：

```
type
  TMyRecord = record
    Value: Integer;
    class operator Initialize (out Dest: TMyRecord);
    class operator Finalize (var Dest: TMyRecord);
  end;
```

请记住，您需要为这两个类方法编写代码。例如，在记录它们的执行或初始化记录值时，我们还记录了对内存位置的引用，以查看哪个记录正在执行每个单独的操作：

```
class operator TMyRecord.Initialize (out Dest: TMyRecord);
begin
  Dest.Value := 10;
  Log('created' + IntToHex (IntPtr(@Dest)));
end;
class operator TMyRecord.Finalize (var Dest: TMyRecord);
begin
  Log('destroyed' + IntToHex (IntPtr(@Dest)));
end;
```

这种构造与以前可用于记录的构造之间的巨大区别在于自动调用。如果编写类似于下面的

代码，则可以同时调用初始值设定项和终结器，并最终得到编译器为托管记录实例生成的 try-finally 块。

```
procedure LocalVarTest;
var
   my1: TMyRecord;begin
   Log (my1.Value.ToString);
end;
```

使用此代码，您将获得如下日志：

```
created 0019F2A8
10
destroyed 0019F2A8
```

另一种情况是使用内联变量，例如：

```
begin
   var t: TMyRecord;
   Log(t.Value.ToString);
```

这会在日志中为您提供相同的输出。

赋值运算符

:= 运算符复制记录字段的所有数据。虽然这是一个合理的默认值，但当您具有自定义数据字段和自定义初始化时，您可能希望更改此行为。这就是为什么对于自定义管理记录，您还可以定义赋值运算符。new 运算符使用:= 运算符语法调用，但定义为赋值运算符：

```
type
   TMyRecord = record
      Value: Integer;
      class operator Assign (var Dest: TMyRecord;
         const [ref] Src: TMyRecord);
```

运算符定义必须遵循非常精确的规则，包括将第一个参数作为引用参数，将第二个参数作为通过引用传递的 const。如果不这样做，编译器将发出如下错误消息：

[dcc32 Error] E2617 First parameter of Assign operator must be a var parameter of the container type

[dcc32 Hint] H2618 Second parameter of Assign operator must be a const[Ref] or var parameter of the container type

有一个调用 赋值运算符的示例案例：

```
var
   my1, my2: TMyRecord;
begin
   my1.Value := 22;
   my2 := my1;
```

生成如下日志（其中序列号包含在记录中）：

```
created 5 0019F2A0
created 6 0019F298
5 copied to 6
destroyed 6 0019F298
destroyed 5 0019F2A0
```

请注意，销毁顺序与构造顺序相反。

赋值运算符与赋值操作结合使用，如上例所示，如果您使用赋值初始化行内变量，也可以使用赋值。这里有两种不同的情况：

```
var
   my1: TMyRecord;begin
   var t := my1;
   Log(t.Value.ToString);

   var s: TMyRecord;
   Log(s.Value.ToString);
```

打印如下日志：

```
created 6 0019F2A8
created 7 0019F2A0
6 copied to 7
10
created 8 0019F298
10
destroyed 8 0019F298
destroyed 7 0019F2A0
destroyed 6 0019F2A8
```

在第一种情况下，创建和赋值的发生方式与使用非局部变量的常规场景类似。在第二种情况下，只有一个常规初始化。

传参差异

托管记录的工作方式与常规记录不同，当作为参数传递或由函数返回时也是如此。下面是显示各种方案的几个例程：

```
procedure ParByValue (rec: TMyRecord);
procedure ParByConstValue (const rec: TMyRecord);
procedure ParByRef (var rec: TMyRecord);
procedure ParByConstRef (const [ref] rec: TMyRecord);
function ParReturned: TMyRecord;
```

每个执行以下操作：

- ✓ ParByValue 创建一个新记录并调用赋值运算符（如果可用）来复制数据，从而在退出过程时销毁临时副本。
- ✓ ParByConstValue 不做复制，不做调用。
- ✓ ParByRef 不做复制，不做调用。
- ✓ ParByConstRef 不做复制，不做调用。
- ✓ ParReturn 创建一个新记录（通过初始化），并在返回时调用 Assign 运算符（如果调用如下所示），并删除临时记录：
  my1 := ParReturned;

异常自动清理

引发异常时，即使没有显式 try, finally 语句，通常也会清除记录，类对象则不然。这是管理记录真正有用的根本区别和关键。

```
procedure ExceptionTest;
begin
   var a: TMRE;
   var b: TMRE;

   raise Exception.Create('Error Message');
end;
```

在此过程中，有两个构造函数调用和两个析构函数调用。同样，这是托管记录的根本区别和关键功能。请参阅后面有关基于托管记录的简单智能指针的部分。

另一方面，如果在托管记录的初始值设定项中引发异常，则不会调用匹配析构函数，这与常规对象发生的情况不同。

记录数组

如果定义托管记录的静态数组，则在点声明处调用 Initialize 运算符进行初始化：

```
var
   a1: array [1..5] of TMyRecord; // call here
begin
   Log ('ArrOfRec');
```

当它们超出范围时，它们都会被销毁。如果定义托管记录的动态数组，则使用数组大小（使用 SetLength）调用初始化代码：

```
var
   a2: array of TMyRecord;
begin
   Log ('ArrOfDyn');
   SetLength(a2, 5); // call here
```

文件类型（Win32）

Win32 平台上提供的文件类型是相同类型的元素序列。标准 I/O 例程使用预定义的 TextFile 或 Text 类型，它表示包含组织成行的字符的文件。有关文件输入和输出的详细信息，请参阅“文件输入和输出”部分下的标准例程和输入输出。

若要声明文件类型，请使用以下语法：

```
type fileTypeName = file of type
```

其中 fileTypeName 是任何有效的标识符，type 是固定大小的类型。不允许使用指针类型（无论是隐式还是显式），因此文件不能包含动态数组、长字符串、类、对象、指针、变体、其他文件或包含其中任何一种的结构化类型。

例如：

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

声明用于记录姓名和电话号码的文件类型。

您也可以使用 file of ... 直接在变量声明中构造。例如

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

单词 file 本身表示非类型化文件：

```
var DataFile: file;
```

有关详细信息，请参阅标准例程和输入输出中的“非类型化文件”。
不允许在数组或记录中使用文件。

指针

指针是表示内存地址的变量。当指针保存另一个变量的地址时，我们说它指向该变量在内存中的位置或存储在那里的数据。对于数组或其他结构化类型，指针保存结构中第一个元素的地址。如果该地址已被占用，则指针将保留指向第一个元素的地址。

键入指针以指示存储在其持有的地址上的数据类型。通用指针类型可以表示指向任何数据的指针，而更专用的指针类型仅引用特定类型的数据。PByte 类型用于任何不是字符数据的字节数据。

在 32 位平台上，指针占用 4 个字节的内存作为 32 位地址。
在 64 位平台上，指针占用 8 个字节的内存作为 64 位地址。

本主题包含有关以下内容的信息：

- ✓ 指针类型的一般概述。
- ✓ 声明和使用 Delphi 支持的指针类型。

指针概述

若要了解指针的工作原理，请查看以下示例：

```
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;
```

第 2 行将 X 和 Y 声明为整数类型的变量。
第 3 行将 P 声明为指向整数值的指针;这意味着 P 可以指向 X 或 Y 的位置.
第 5 行为 X 分配一个值，
第 6 行将 X 的地址（用 @X 表示）分配给 P。最后，
第 7 行检索 P 指向的位置（用 ^P 表示）的值并将其分配给 Y。
执行此代码后，X 和 Y 具有相同的值，即 17。

@ 运算符（此处用于获取变量的地址）也对函数和过程进行操作。有关更多信息，请参

见语句和表达式中的 @ 运算符和过程类型。

插入符号 ^ 有两个用途，在我们的示例中都说明了这两个用途。当它出现在类型标识符之前时：

```
^typeName
```

插入符号表示一个类型，该类型表示指向类型为 typeName 的变量的指针。

当插入符号出现在指针变量之后时：

```
pointer^
```

插入符号取消引用指针;也就是说，它返回存储在指针持有的内存地址处的值。

这个例子可能看起来像是将一个变量的值复制到另一个变量的迂回方式 - 我们可以通过一个简单的赋值语句来完成。但是指针很有用有几个原因。**首先，**理解指针将帮助您理解 Delphi 语言，因为指针通常在幕后代码中运行，它们不会显式出现。任何需要动态分配的大型内存块的数据类型都使用指针。例如，长字符串变量是隐式指针，类实例变量也是如此。**此外，**一些高级编程技术需要使用指针。

最后，指针有时是规避 Delphi 严格数据类型的唯一方法。通过使用通用指针引用变量，将指针强制转换为更具体的类型，然后取消引用它，可以将任何变量存储的数据视为属于任何类型。例如，以下代码将存储在实变量中的数据分配给整数变量：

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

当然，实数和整数以不同的格式存储。此赋值只是将原始二进制数据从 R 复制到 I，而不对其进行转换。

除了分配 @ 操作的结果之外，还可以使用多个标准例程为指针赋值。New 和 GetMem 过程将内存地址分配给现有指针，而 Addr 和 Ptr 函数返回指向指定地址或变量的指针。

取消引用的指针可以是限定的，并且可以用作限定符，如表达式 P1^.Data^所示。

保留字 nil 是一个特殊的常量，可以分配给任何指针。将 nil 分配给指针时，指针不引用任何内容。

将扩展语法与指针结合使用

{$EXTENDED} 编译器指令会影响插入符号 （^） 的使用。当 {$X+} 生效（默认值）时，可以在引用指针时省略插入符号。在声明指针时，以及当指针指向另一个指针时解决歧义时，仍然需要插入符号。有关详细信息，请参阅编译器指令中的扩展语法 （Delphi）。

启用扩展语法后，可以在引用指针时省略插入符号，如以下示例所示：

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

 var
    MyRec: PMyRec;

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

如果未启用扩展语法，标记为 {#1} 的行通常表示为：

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

指针类型

可以使用以下语法声明指向任何类型的指针：

```
type pointerTypeName = ^type
```

定义记录或其他数据类型时，定义指向该类型的指针可能很有用。这使得操作该类型的实例变得容易，而无需复制大内存块。

注意： 可以在声明指针类型指向的类型之前声明指针类型。

标准指针类型有多种用途。最通用的是 Pointer，它可以指向任何类型的数据。但是指针变量不能取消引用;将 ^ 符号放在指针变量之后会导致编译错误。若要访问 Pointer 变量引用的数据，请先将其强制转换为另一个指针类型，然后取消引用它。

字符指针

基本类型 PAnsiChar 和 PWideChar 分别表示指向 AnsiChar 和 WideChar 值的指针。通

用 PChar 表示指向 Char 的指针（即，在其当前实现中，指向 WideChar）。这些字符指针用于操作以 null 结尾的字符串。（请参阅字符串类型 （Delphi） 中的《“使用以 null 结尾的字符串”》。

注意：不要将非字符指针类型强制转换为 PChar 以执行指针算术。请改用 PByte 指针类型，该类型使用 {$POINTERMATH ON} 编译器指令声明。

字节指针

基本类型 PByte 表示指向任何非字符数据的字节数据的指针。此类型使用 {$POINTERMATH ON} 编译器指令声明：

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

类型检查指针

$T 编译器指令控制 @ 运算符生成的指针值的类型。该指令采用以下形式：

{$T+} or {$T-}

在 {$T-} 状态下，@ 运算符的结果类型始终是与所有其他指针类型兼容的非类型化指针。当 @ 应用于 {$T+} 状态下的变量引用时，结果的类型为 ^T，其中 T 仅与指向变量类型的指针兼容。

其他标准指针类型

系统和系统实用程序单元声明了许多常用的标准指针类型。

使用 {POINTERMATH <ON|OFF>} 指令为所有类型化指针打开或关闭指针算术，以便按元素大小递增/递减。

在系统和系统实用程序中声明的选定指针类型：

指针类型	指向的变量数据类型
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.

过程类型

过程类型允许您将过程和函数视为可分配给变量或传递给其他过程和函数的值。

本主题不引用与匿名方法一起使用的较新的过程类型，即“对过程的引用”。请参阅《Delphi中的匿名方法》。

关于过程类型

下面的示例演示过程类型的用法。假设您定义一个名为 Calc 的函数，该函数接受两个整数参数并返回一个整数：

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

您可以将 Calc 函数分配给变量 F：

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

如果采用任何过程或函数标题并删除单词过程或函数后面的标识符，则剩下的是过程类型声明的类型名称部分。可以直接在变量声明中使用此类类型名称（如前面的示例所示）或声明新类型：

```
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
```

方法指针

上一示例中所示的变量都是过程指针，即指向过程或函数地址的指针。如果要引用实例对象的方法（请参阅类和对象（Delphi）），则需要将 of object 添加到过程类型名称中。例如：

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

这些类型表示方法指针。方法指针实际上是一对指针;第一个存储方法的地址，第二个存储对方法所属对象的引用。鉴于声明：

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

我们可以进行以下分配：

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

如果两种过程类型具有以下条件，则它们兼容：

- ✓ 相同的调用约定，
- ✓ 相同的返回值（或无返回值），以及
- ✓ 相同数量的参数，在相应位置具有相同类型的参数。（参数名称无关紧要。)

过程指针类型始终与方法指针类型不兼容。值 nil 可以分配给任何过程类型。

嵌套过程和函数（在其他例程中声明的例程）不能用作过程值，预定义的过程和函数也不能用作过程值。如果要使用预定义的例程（如 Length）作为过程值，请为其编写包装器：

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

用于语句和表达式

当过程变量位于赋值语句的左侧时，编译器期望过程值位于右侧。赋值使左侧的变量成为指向右侧指示的函数或过程的指针。但是，在其他上下文中，使用过程变量会导致调用引用的过程或函数。您甚至可以使用过程变量来传递参数：

```
  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
```

在赋值语句中，左侧变量的类型决定了右侧过程或方法指针的解释。例如：

```
  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
```

第一条语句将过程值赋给 F。第二条语句将该值复制到另一个变量。第三条语句调用引用的函数并将结果分配给 I。因为 I 是一个整数变量，而不是过程变量，所以最后一个赋值实际上调用了函数（返回一个整数）。

在某些情况下，如何解释程序变量不太清楚。考虑以下语句：

```
if F = MyFunction then ...;
```

在这种情况下，F 的出现会导致函数调用;编译器调用 F 指向的函数，然后调用函数

MyFunction，然后比较结果。规则是，每当表达式中出现过程变量时，它表示对引用的过程或函数的调用。在 F 引用过程（不返回值）或 F 引用需要参数的函数的情况下，前面的语句会导致编译错误。要将 F 的过程值与 MyFunction 进行比较，请使用：

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

@F 将 F 转换为包含地址的非类型指针变量，@MyFunction 返回 MyFunction 的地址。

若要获取过程变量的内存地址（而不是存储在其中的地址），请使用 @@。例如，@@F 返回 F 的地址。

@ 运算符还可用于将非类型化指针值分配给过程变量。例如：

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

调用 GetProcAddress 函数并将 StrComp 指向结果。

任何过程变量都可以保存值 nil，这意味着它不指向任何内容。但是尝试调用零值过程变量是一个错误。要测试是否分配了过程变量，请使用标准函数 已分配：

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

变体

变体概述

有时需要操作类型变化或在编译时无法确定的数据。在这些情况下，一个选项是使用 Variant 类型的变量和参数，它们表示可以在运行时更改类型的值。变体提供了更大的灵活性，但比常规变量消耗更多的内存，并且对它们的操作比对静态绑定类型的操作慢。此外，对变体的非法操作通常会导致运行时错误，其中常规变量的类似错误会在编译时被捕获。您还可以创建自定义变体类型。

默认情况下，Variant 可以保存任何类型的值，但记录、集合、静态数组、文件、类、类引用和指针除外。换句话说，变体可以包含除结构化类型和指针之外的任何内容。它们可以保存接口，可以通过它们访问其方法和属性。（请参阅对象接口 （Delphi）。）它们可以容纳动态数组，并且可以容纳一种称为变体数组的特殊静态数组。（请参阅本章后面的“变体数组”。）变体可以与其他变体以及表达式和赋值中的整数、实数、字符串和布尔值混合;编译器会自动执行类型转换。

无法为包含字符串的变体编制索引。也就是说，如果 V 是保存字符串值的变体，则构造 V[1] 会导致运行时错误。

您可以定义自定义变体，以扩展变体类型以保存任意值。例如，可以定义允许索引或保存特定类引用、记录类型或静态数组的 Variant 字符串类型。自定义变体类型是通过创建 TCustomVariantType 类的后代来定义的。

注意：这和几乎所有的变体功能都是在 System.Variant 单元中实现的。

注意：变体记录本质上被视为“不安全”。变体记录与使用“absolute”指令非常相似，

因为记录的变体字段部分实际上覆盖在内存中。可以将值指定为一种类型，然后将其读出为另一种类型。如果使用变体，则可能会看到有关不安全代码的编译器警告，例如 W1047 Unsafe code '%s' (Delphi)。

在 32 位平台上，变体存储为 16 字节记录。在 64 位平台上，变体存储为 24 字节记录。变体记录由类型代码和类型代码指定的类型的值或指向值的指针组成。所有变体在创建时初始化为特殊值“未分配”。特殊值 Null 表示未知或缺失的数据。

标准函数 VarType 返回变体的类型代码。varTypeMask 常量是一个位掩码，用于从 VarType 的返回值中提取代码，例如，

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

如果 V 包含双精度或双精度数组，则返回 True。（掩码只是隐藏第一位，指示变体是否包含数组。系统单元中定义的 TVarData 记录类型可用于对变体进行类型转换并访问其内部表示形式。

变体类型转换

所有整数、实数、字符串、字符和布尔类型都与 Variant 赋值兼容。表达式可以显式转换为变体，并且 VarAsType 和 VarCast 标准例程可用于更改变体的内部表示形式。以下代码演示了变体的使用以及变体与其他类型的混合时执行的一些自动转换：

```
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;
```

编译器根据以下规则执行类型转换：

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).

超出范围的赋值通常会导致目标变量获得其范围内的最大值。无效的变体操作、赋值或强制转换会引发 Variants.EVariantError 异常或从 Variants.EVariantError 衍生的异常类。

特殊转换规则适用于在系统单元中声明的 System.TDateTime 类型。当 System.TDateTime 转换为任何其他类型时，它被视为普通的双精度。当整数、实数或布尔值转换为

System.TDateTime 时，它首先转换为 Double，然后读取为日期时间值。当字符串转换为 System.TDateTime 时，将使用区域设置将其解释为日期时间值。当“未赋值”转换为 System.TDateTime 时，它被视为实值或整数值 0。将 Null 值转换为 System.TDateTime 会引发异常。

在 Win32 平台上，如果变体引用 COM 接口，则任何转换它的尝试都会读取对象的默认属性，并将该值转换为请求的类型。如果对象没有默认属性，则会引发异常。

用于表达式

除 ^、is 和 in 之外的所有运算符都采用变体操作数。除了始终返回布尔结果的比较之外，对变体值的任何操作都会返回变体结果。如果表达式将变体与静态类型值组合在一起，则静态类型值将自动转换为变体。

对于比较则不然，在比较中，对 Null 变体的任何操作都会生成 Null 变体。例如：

```
V := Null + 3;
```

将 Null 变体分配给 V。默认情况下，比较将 Null 变体视为小于任何其他值的唯一值。例如：

```
if Null > -3 then ... else ...;
```

在此示例中，将执行 if 语句的 else 部分。可以通过设置 NullEqualRule 和 NullMagnitudeRule 全局变量来更改此行为。

变体数组

不能将普通静态数组分配给变体。相反，通过调用标准函数 VarArrayCreate 或 VarArrayOf 中的任何一个来创建变体数组。例如：

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

创建一个整数的变体数组（长度为 10），并将其分配给变量 V。数组可以使用 V[0]、V[1] 等进行索引，但不能将变体数组元素作为 var 参数传递。变体数组始终使用整数进行索引。

调用 VarArrayCreate 中的第二个参数是数组基类型的类型代码。有关这些代码的列表，请参阅 VarType。永远不要将代码 varString 传递给 VarArrayCreate;要创建字符串的变体数组，请使用 varOleStr。

变体可以容纳不同大小、维度和基本类型的变体数组。变体数组的元素可以是变体中允许的任何类型，但 ShortString 和 AnsiString 除外，如果数组的基本类型是 Variant，则其元素甚至可以是异构的。使用 VarArrayRedim 函数调整变体数组的大小。在变体数组上运行的其他标准例程包括 VarArrayDimCount、VarArrayLowBound、VarArrayHighBound、VarArrayRef、VarArrayLock 和 VarArrayUnlock。

注意：不支持自定义变体的变体数组，因为可以将自定义变体的实例添加到 VarVariant 变体数组中。

当包含变体数组的变体分配给另一个变体或作为值参数传递时，将复制整个数组。不要不必要地执行此类操作，因为它们的内存效率低下。

OleVariant

Variant 和 OleVariant 之间的主要区别在于，Variant 可以包含只有当前应用程序知道如何处理的数据类型。OleVariant 只能包含定义为与 OLE 自动化兼容的数据类型，这意味着可以在程序之间或通过网络传递的数据类型，而不必担心另一端是否知道如何处理数据。

将包含自定义数据（如 Delphi 字符串或新的自定义变量类型之一）的变体分配给 OleVariant 时，运行时库会尝试将该变量转换为 OleVariant 标准数据类型之一（如 Delphi 字符串转换为 OLE BSTR 字符串）。例如，如果将包含 AnsiString 的变体分配给 OleVariant，则该 AnsiString 将变为 WideString。将变量传递给 OleVariant 函数参数时也是如此。

类型兼容和等同

要了解可以对哪些表达式执行哪些操作，我们需要区分类型和值之间的几种兼容性。其中包括：

- ✓ 类型身份
- ✓ 类型兼容
- ✓ 赋值兼容

区分类型

当一个类型标识符使用另一个类型标识符声明时，不带限定条件，它们表示相同的类型。因此，鉴于声明：

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

T1、T2、T3、T4 和整数都表示相同的类型。若要创建不同的类型，请在声明中重复单词 type。例如：

```
type TMyInteger = type Integer;
```

创建一个名为 TMyInteger 的新类型，该类型与 Integer 不同。

用作类型名称的语言构造每次出现时都表示不同的类型。因此，声明：

```
type
   TS1 = set of Char;
   TS2 = set of Char;
```

创建两种不同的类型：TS1 和 TS2。同样，变量声明：

```
var
   S1: string[10];
   S2: string[10];
```

创建两个不同类型的变量。要创建相同类型的变量，请使用：

```
var S1, S2: string[10];
```

或：

```
type MyString = string[10];
var
      S1: MyString;
      S2: MyString;
```

类型兼容

每种类型都与自身兼容。如果两种不同的类型至少满足以下条件之一，则它们兼容。

- ✓ 它们都是浮点类型。
- ✓ 它们都是整数类型。
- ✓ 一种类型是另一种类型的子范围。
- ✓ 这两种类型都是同一类型的子范围。
- ✓ 两者都是具有相互兼容基本类型的集合类型。
- ✓ 两者都是具有相同字符数的 packed 字符串类型。
- ✓ 一个是字符串类型，另一个是字符串、packed 字符串或 Char 类型。
- ✓ 一种类型是 Variant 类型，另一种是整数、实数、字符串、字符或布尔类型。
- ✓ 两者都是类、类引用或接口类型，一种类型派生自另一种类型。
- ✓ 一种类型是 PAnsiChar 或 PWideChar，另一种是从零开始的字符数组，形式为 array[0..n] of PAnsiChar or PWideChar。
- ✓ 一种类型是指针（非类型化指针），另一种是任何指针类型。
- ✓ 这两种类型都是指向同一类型的（类型化）指针，并且 {$T+} 编译器指令有效。
- ✓ 两者都是过程类型，具有相同的结果类型、相同数量的参数以及相应位置的参数之间的类型标识。

赋值兼容

赋值兼容性不是一种对称关系。如果 T2 类型的表达式的值在 T1 范围内，并且至少满足以下条件之一，则可以将 T2 类型的表达式分配给该变量 T1 ：

- ✓ T1 和 T2 属于同一类型，它不是文件类型或包含任何级别的文件类型的结构化类型。
- ✓ T1 和 T2 是兼容的序号类型。
- ✓ T1 和 T2 都是实数类型。
- ✓ T1 是实数类型，T2 是整数类型。
- ✓ T1 是 PAnsiChar、PWideChar、PChar 或任何字符串类型，表达式是字符串常量。
- ✓ T1 和 T2 都是字符串类型。
- ✓ T1 是字符串类型，T2 是字符或 packed 字符串类型。
- ✓ T1 是一个长字符串，T2 是 PAnsiChar、PWideChar 或 PChar。
- ✓ T1 和 T2 是兼容的 packed 字符串类型。
- ✓ T1 和 T2 是兼容的集合类型。
- ✓ T1 和 T2 是兼容的指针类型。
- ✓ T1 和 T2 都是类、类引用或接口类型，T2 是从 T1 派生的。
- ✓ T1 是接口类型，T2 是实现 T1 的类类型。
- ✓ T1 是 PAnsiChar 或 PWideChar，T2 是从零开始的字符数组，其形式数组 array[0..n] of Char（当 T1 为 PAnsiChar 时）或 WideChar（当 T1 为 PWideChar 时）。
- ✓ T1 和 T2 是兼容的过程类型。（在某些赋值语句中，函数或过程标识符被视为过程类型的表达式。请参阅本章前面的“语句和表达式中的过程类型”。
- ✓ T1 是变体，T2 是整数、实数、字符串、字符、布尔值、接口类型或 OleVariant 类型。
- ✓ T1 是 OleVariant ，T2 是整数、实数、字符串、字符、布尔值、接口或变体类型。
- ✓ T1 是整数、实数、字符串、字符或布尔类型，T2 是变体或 OleVariant 。
- ✓ T1 是 IUnknown 或 IDispatch 接口类型，T2 是 Variant 或 OleVariant。（如果 T1 是 IUnknown，则变体的类型代码必须为 varEmpty、varUnknown 或 varDispatch，如果 T1 是 IDispatch，则为 varEmpty 或 varDispatch。

声明类型

类型声明指定表示类型的标识符。类型声明的语法为：

```
type newTypeName = type
```

其中 newTypeName 是有效的标识符。例如，给定类型声明：

```
type TMyString = string;
```

您可以进行变量声明：

```
var S: TMyString;
```

类型标识符的作用域不包括类型声明本身（指针类型除外）。因此，例如，您不能定义以递归方式使用自身的记录类型。

声明与现有类型相同的类型时，编译器会将新类型标识符视为旧类型标识符的别名。因此，鉴于声明：

```
type TValue = Real;
var
   X: Real;
   Y: TValue;
```

X 和 Y 属于同一类型;在运行时，无法区分 TValue 和 Real。这通常无关紧要，但如果定义新类型的目的是利用运行时类型信息，例如，将属性编辑器与特定类型的属性相关联，则“不同名称”和“不同类型”之间的区别变得很重要。在这种情况下，请使用以下语法：

```
type newTypeName = type KnownType
```

例如：

```
type TValue = type Real;
```

强制编译器创建一个名为 TValue 的新的独特类型。
对于 var 参数，形参和实参的类型必须相同。例如：

```
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;
```

注意：这仅适用于 var 参数，不适用于常量或按值参数。

声明变量

变量是其值可以在运行时更改的标识符。换句话说，变量是内存中某个位置的名称;可以使用该名称读取或写入内存位置。变量就像数据的容器，因为它们是强类型的，所以它们告诉

编译器如何解释它们所保存的数据。

声明语法

变量声明的基本语法为：

```
var identifierList:type;
```

其中 identifierList 是逗号分隔的有效标识符列表，类型是任何有效类型。例如：

```
var I: Integer;
```

声明一个整数类型的变量 I，同时：

```
var X, Y: Real;
```

声明两个变量 - X 和 Y - 类型为 Real。
连续变量声明不必重复输入保留字 var：

```
var
    X, Y, Z: Double;
    I, J, K: Integer;
    Digit: 0..9;
    Okay: Boolean;
```

在过程或函数中声明的变量有时称为局部变量，而其他变量称为全局变量。全局变量可以在声明的同时初始化，使用以下语法：

```
var identifier: type = constantExpression;
```

其中常量表达式是表示类型类型的值的任何常量表达式。因此声明：

```
var I: Integer = 7;
```

等效于声明和语句：

```
var I: Integer;
      ...
I := 7;
```

局部变量不能在其声明中初始化。多个变量声明（如 var X， Y， Z： REAL;） 不能包含初始化，变体和文件类型变量的声明也不能包含初始化。
如果未显式初始化全局变量，编译器会将其初始化为 0。对象实例数据（字段）也初始化为 0。在为局部变量赋值之前，局部变量的内容是未定义的。

当您声明一个变量时，您正在分配内存，当不再使用该变量时，该内存会自动释放。特别是，局部变量仅在程序退出声明它们的函数或过程之前存在。有关变量和内存管理的详细信息，请参阅内存管理。

绝对地址变量

您可以创建一个与另一个变量位于同一地址的新变量。为此，请将指令 absolute 放在新变量声明中的类型名称之后，后跟现有（以前声明的）变量的名称。例如：

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

指定变量 StrLen 应从与 Str 相同的地址开始。由于短字符串的第一个字节包含字符串长度，因此 StrLen 的值是 Str 的长度。

不能初始化绝对声明中的变量，也不能将绝对与任何其他指令组合。

动态变量

可以通过调用 GetMem 或 New 过程来创建动态变量。此类变量在堆上分配，不会自动管理。创建变量后，您最终有责任释放变量的内存;使用 FreeMem 销毁 GetMem 创建的变量，使用 Dispose 销毁 New 创建的变量。其他对动态变量进行操作的标准例程包括 ReallocMem，AllocMem，Initialize，Finalize，StrAlloc 和 StrDispose。

长字符串、宽字符串、动态数组、变体和接口也是堆分配的动态变量，但它们的内存是自动管理的。

线程局部变量

线程局部（或线程）变量用于多线程应用程序。线程局部变量类似于全局变量，不同之处在于每个执行线程都获得自己的变量私有副本，无法从其他线程访问该副本。线程局部变量是用 threadvar 而不是 var 声明的。例如：

```
threadvar X: Integer;
```

线程变量声明：

- ✓ 不能在过程或函数中出现。
- ✓ 不能包含初始化。
- ✓ 不能指定绝对指令。

通常由编译器管理的动态变量（长字符串、宽字符串、动态数组、变体和接口）可以使用 threadvar 声明，但编译器不会自动释放由每个执行线程创建的堆分配内存。如果在线程变量

中使用这些数据类型，则您有责任在线程终止之前从线程内部释放它们的内存。例如：

```
threadvar S: AnsiString;
  S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
     ...
  S := '';    // free the memory used by S
```

注意：不鼓励使用此类结构。

您可以通过将变体设置为“未分配”来释放变体，也可以通过将接口或动态数组设置为 nil 来释放它。

声明常量

几种不同的语言结构被称为“常量”。有数字常量（也称为数字），如 17，以及字符串常量（也称为字符串或字符串文字），如“Hello world！每个枚举类型都定义表示该类型值的常量。有预定义的常量，如 True、False 和 nil。最后，有些常量与变量一样，是通过声明单独创建的。

声明的常量是普通常量或类型化常量。这两种常量表面上相似，但它们受不同的规则支配，用于不同的目的。

普通常量声明

普通常量是声明的标识符，其值不能更改。例如：

```
const MaxValue = 237;
```

声明一个名为 MaxValue 的常量，该常量返回整数 237。声明普通常量的语法为：

```
const identifier = constantExpression
```

其中 identifier 是任何有效的标识符，constantExpression 是编译器无需执行程序即可计算的表达式。

如果 constantExpression 返回序号值，则可以使用值类型转换指定声明常量的类型。例如：

```
const MyNumber = Int64(17);
```

声明一个名为 MyNumber 的常量，类型为 Int64，返回整数 17。否则，声明的常量的类型是常量表达式的类型。

如果 constantExpression 是字符串，则声明的常量与任何字符串类型兼容。如果字符串的长度为 1，则它也与任何字符类型兼容。

如果 constantExpression 是实数，则其类型为 Extended。如果是整数，则其类型由下表给出：

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 and Android.

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

$FFFFFFFF	4294967295		

64-bit platforms integer type excluding 64-bit Windows
64-bit platforms include 64-bit iOS, 64-bit Android, 64-bit macOS and 64-bit Linux.

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

下面是常量声明的一些示例：

```
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;
```

常量表达式

常量表达式是编译器无需执行发生常量表达式的程序即可计算的表达式。常量表达式包括数字;字符串;普通常量;枚举类型的值;特殊常数 true、false 和 nil;以及专门从这些元素构建的表达式，其中包含运算符、类型转换和集合构造函数。常量表达式不能包含变量、指针或函数调用，但对以下预定义函数的调用除外：

Abs	High	Low	Pred	Succ
Chr	Length	Odd	Round	Swap
Hi	Lo	Ord	SizeOf	Trunc

常量表达式的这个定义在 Delphi 语法规范中的多个地方使用。常量表达式是初始化全局变量、定义子范围类型、为枚举类型中的值分配序数、指定默认参数值、编写 case 语句以及声明 普通常量和类型常量所必需的。

常量表达式的示例：

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Embarcadero' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

资源常量字符串

资源字符串存储为资源并链接到可执行文件或库中，以便无需重新编译程序即可对其进行修改。

资源字符串声明为其他普通常量，只是单词 const 被 resourcestring 替换。= 符号右侧的表达式必须是常量表达式，并且必须返回字符串值。例如：

```
resourcestring
  CreateError = 'Cannot create file %s';
  OpenError = 'Cannot open file %s';
  LineTooLong = 'Line too long';
  ProductName = 'Embarcadero Rocks';
  SomeResourceString = SomeTrueConstant;
```

类型化常量

与 普通常量不同，类型化常量可以保存数组、记录、过程和指针类型的值。类型化常量不能出现在常量表达式中。

声明一个类型常量，如下所示：

```
const identifier: type = value
```

其中，identifier 是任何有效的标识符，type 是除文件和变体之外的任何类型，value 是类型的表达式。例如

```
const Max: Integer = 100;
```

在大多数情况下，值必须是常量表达式;但是，如果类型是数组、记录、过程或指针类型，则适用特殊规则。

数组常量

若要声明数组常量，请将数组元素的值（用逗号分隔）括在声明末尾的括号中。这些值必

须由常量表达式表示。例如：

```
const Digits: array[0..9] of Char =
  ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

声明一个名为 Digits 的类型化常量，该常量保存一个字符数组。

从零开始的字符数组通常表示以 null 结尾的字符串，因此可以使用字符串常量来初始化字符数组。因此，前面的声明可以更方便地表示为：

```
const Digits: array[0..9] of Char = '0123456789';
```

若要定义多维数组常量，请将每个维度的值括在一组单独的括号中，用逗号分隔。例如：

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

创建一个名为 Maze 的数组，其中：

```
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
```

数组常量不能包含任何级别的文件类型值。

记录常量

若要声明记录常量，请在声明末尾的括号中指定每个字段的值 - 作为 fieldName： value，字段赋值用分号分隔。这些值必须由常量表达式表示。字段必须按照它们在记录类型声明中出现的顺序列出，并且标记字段（如果有）必须指定值;如果记录具有变型部件，则只能为标记字段选择的变型分配值。

例子：

```
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);
```

记录常量不能包含任何级别的文件类型值。

过程常量

若要声明过程常量，请指定与常量声明的类型兼容的函数或过程的名称。例如

```
function Calc(X, Y: Integer): Integer;
begin
  ...
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

给定这些声明，您可以在函数调用中使用过程常量 MyFunction：

```
I := MyFunction(5, 7)
```

还可以将值 nil 赋值给过程常量。

指针常量

声明指针常量时，必须将其初始化为至少可以在编译时解析为相对地址的值。有三种方法可以执行此操作：使用 @ 运算符、使用 nil 和（如果常量类型为 PChar 或 PWideChar）使用字符串文本。例如，如果 I 是 Integer 类型的全局变量，则可以声明一个常量，如下所示：

```
const PI: ^Integer = @I;
```

编译器可以解决此问题，因为全局变量是代码段的一部分。函数和全局常量也是如此：

```
const PF: Pointer = @MyFunction;
```

由于字符串文本被分配为全局常量，因此可以使用字符串文本初始化 PChar 常量：

```
const WarningStr: PChar = 'Warning!';
```

可修改类型常量

Delphi 允许修改类型化常量，如果您设置编译器指令 （$J+} 或可写类型化常量 （Delphi） {$WRITEABLECONST ON}。

设置 $J+ 后，可以使用赋值语句更改类型化常量的值，就像它们是初始化的变量一样。例如：

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

可写类型化常量和初始化变量之间的差异：

- ✓ 可写类型化常量可以在过程、函数和方法中全局和局部出现。
- ✓ 初始化的变量只能用作全局声明。
- ✓ 初始化的变量在过程或方法中尝试时会导致编译时错误。

5. 过程和函数

过程和函数

概述

过程和函数统称为例程，是可以从程序中的不同位置调用的独立语句块。函数是在执行时返回值的例程。过程是不返回值的例程。

函数调用（因为它们返回一个值）可以用作赋值和操作中的表达式。例如：

```
I := SomeFunction(X);
```

调用 SomeFunction 并将结果分配给 I.函数调用不能出现在赋值语句的左侧。

过程调用 - 以及启用扩展语法 （{$X+}） 时，函数调用 - 可以用作完整语句。例如：

```
DoSomething;
```

调用 DoSomething 例程;如果 DoSomething 是一个函数，则丢弃其返回值。
过程和函数可以递归调用自己。

声明语法

声明过程或函数时，指定其名称、它采用的参数的数量和类型，如果是函数，则指定其返回值的类型;声明的这一部分有时称为原型、标题或头部。然后，您编写一个代码块，每当调用过程或函数时都会执行;这部分有时称为例程或块的主体。

过程声明

过程声明的格式为：

```
procedure procedureName(parameterList); directives;
   localDeclarations;
begin
   statements
end;
```

其中 procedureName 是任何有效的标识符，statements 是在调用过程时执行的语句序列，以及 （parameterList）、directives 和 localDeclarations;是可选的。
下面是过程声明的示例：

```
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;
```

给定此声明，可以像这样调用 NumString 过程：

```
NumString(17, MyString);
```

此过程调用将值“17”分配给 MyString（必须是字符串变量）。
在过程的语句块中，可以使用在过程的 localDeclarations 部分中声明的变量和其他标识符。您还可以使用参数列表中的参数名称（如上一示例中的 N 和 S）;参数列表定义了一组局部变

量，因此不要尝试在 localDeclarations 部分中重新声明参数名称。最后，可以使用过程声明所属范围的任何标识符。

函数声明

函数声明类似于过程声明，不同之处在于它指定返回类型和返回值。函数声明的格式为：

```
function functionName(parameterList): returnType; directives;
  localDeclarations;
begin
  statements
end;
```

其中 functionName 是任何有效的标识符，returnType 是类型标识符，statements 是调用函数时执行的语句序列，以及 （parameterList）、directives 和 localDeclarations;是可选的。

函数的语句块受适用于过程的相同规则的约束。在语句块中，可以使用在函数的 localDeclarations 部分中声明的变量和其他标识符、参数列表中的参数名称以及函数声明所属范围的任何标识符。此外，函数名称本身充当保存函数返回值的特殊变量，预定义变量 Result 也是如此。

只要启用了扩展语法 （{$X+}），就会在每个函数中隐式声明 Result。不要试图重新声明它。

例如：

```
function WF: Integer;
begin
  WF := 17;
end;
```

定义一个名为 WF 的常量函数，该函数不带任何参数，并且始终返回整数值 17。此声明等效于：

```
function WF: Integer;
begin
  Result := 17;
end;
```

下面是一个更复杂的函数声明：

```
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;
```

可以在语句块中重复为 Result 或函数名称赋值，只要只赋值与声明的返回类型匹配即可。当函数的执行终止时，上次分配给 Result 或函数名称的任何值都将成为函数的返回值。例如：

```
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;
```

结果和函数名称始终表示相同的值。因此：

```
function MyFunction: Integer;
begin
  MyFunction := 5;
  Result := Result * 2;
  MyFunction := Result + 1;
end;
```

返回值 11。但结果不能与函数名称完全互换。当函数名称出现在赋值语句的左侧时，编译器假定它正在用于（如 Result）来跟踪返回值;当函数名称出现在语句块中的其他任何位置时，编译器会将其解释为对函数本身的递归调用。另一方面，结果可以用作操作、类型转换、集合构造函数、索引和调用其他例程中的变量。

如果函数退出时没有为 Result 赋值或函数名称，则函数的返回值是未定义的。

调用约定

声明过程或函数时，可以使用以下指令之一指定调用约定：register、pascal、cdecl、stdcall、safecall 和 winapi。例如：

```
function MyFunction(X, Y: Real): Real; cdecl;
```

调用约定确定将参数传递给例程的顺序。它们还会影响从堆栈中删除参数、使用寄存器传递参数以及错误和异常处理。默认调用约定为 register。

- ✓ 对于 register 和 pacal 约定，未定义求值顺序。
- ✓ cdecl、stdcall 和 safecall 约定从右到左传递参数。
- ✓ 对于除 cdecl 之外的所有约定，过程或函数在返回时从堆栈中删除参数。使用 cdecl 约定，调用方在调用返回时从堆栈中删除参数。
- ✓ register 约定最多使用三个 CPU 寄存器来传递参数，而其他约定则传递堆栈上的所有参数。
- ✓ safecall 约定实现异常“防火墙”。在 Win32 上，这实现了进程间 COM 错误通知。
- ✓ winapi 实际上不是一个调用约定。winapi 使用默认平台调用约定进行定义。例如，在 Win32 上，winapi 与 stdcall 相同。

下表总结了调用约定:

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

默认 register 约定是最有效的，因为它通常避免创建堆栈帧。（已发布属性的访问方法必须使用 register 。当您从用 C 或 C++ 编写的共享库中调用函数时，cdecl 约定很有用，而通常建议使用 stdcall 和 safecall 来调用外部代码。在 Win32 上，操作系统 API 是 stdcall 和 safecall。其他操作系统通常使用 cdecl。（请注意，stdcall 比 cdecl 更有效。）

必须使用 safecall 约定来声明双接口方法。保留 pascal 约定是为了向后兼容。

near, far, export 是指 16 位 Windows 编程中的调用约定。它们对任何当前平台目标都没有影响，并且仅出于向后兼容性而维护。

前向声明

forward 指令替换过程或函数声明中的块，包括局部变量声明和语句。例如：

```
function Calculate(X, Y: Integer): Real; forward;
```

声明一个名为“Calculate”的函数。在前向声明之后的某个地方，必须在包含块的定义声明中重新声明例程。Calculate 的定义声明可能如下所示：

```
function Calculate;
  ... { declarations }
begin
  ... { statement block }
```

```
end;
```

通常，定义声明不必重复例程的参数列表或返回类型，但如果确实重复了它们，则它们必须与前向声明中的参数完全匹配（除了可以省略默认参数）。如果前向声明指定了重载过程或函数，则定义声明必须重复参数列表。

前向声明及其定义声明必须出现在同一类型声明节中。也就是说，不能在前向声明和定义声明之间添加新节（如 var 节或 const 节）。定义声明可以是外部声明或汇编程序声明，但不能是另一个前向声明。

前向声明的目的是将过程或函数标识符的范围扩展到源代码中的较早点。这允许其他过程和函数在实际定义前向声明的例程之前调用它。除了让您更灵活地组织代码之外，前向声明有时对于相互递归也是必需的。

前向指令在单元的接口部分中无效。接口部分中的过程和函数标头的行为类似于前向声明，并且必须在实现部分中具有定义声明。接口部分中声明的例程可从单元中的其他任何位置以及使用声明该例程的单元的任何其他单元或程序使用。

外部声明

external 指令替换过程或函数声明中的块，允许您调用与程序分开编译的例程。外部例程可以来自.obj 文件或可动态加载的库。

导入采用可变数量参数的 C 函数时，请使用 varargs 指令。例如：

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

varargs 指令仅适用于外部例程，并且仅适用于 cdecl 调用约定。

链接.obj 文件

若要从单独编译的对象文件调用例程，请首先使用 $L（或 $LINK）编译器指令将对象文件链接到应用程序。例如：

```
{$L BLOCK.OBJ}
```

链接 BLOCK.OBJ 进入它发生的程序或单元。接下来，声明要调用的函数和过程：

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

现在，您可以从 BLOCK.OBJ 调用 MoveWord 和 FillWord 例程。

在 Win32 平台上，像上面这样的声明经常用于访问用汇编语言编写的外部例程。您还可以将汇编语言例程直接放置在 Delphi 源代码中。

从外部库导入函数

要从可动态加载的库 （.DLL） 导入例程，请附加以下形式的指令：

external stringConstant;

到正常过程或函数标头的末尾，其中 stringConstant 是单引号中的库文件的名称。例如，在 32 位 Windows 上

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

从 strlib.dll 导入一个名为 SomeFunction 的函数。

使用内部和外部连接器

Delphi 对 external 有两种解释，这取决于编译器是否使用外部链接器
1. Delphi 支持的平台可分为以下两类：

- ✓ 编译器使用自己的内部链接器。
- ✓ 编译器使用外部链接器的位置。

Using internal linker	WIN32 and WIN64;
Using external linker	iOS-Device, Android, Linux, macOS64

2. 在 Delphi 使用内部链接器的平台上，外部<stringconstant>表示函数/过程位于 DLL 中。
在这些平台上，Delphi 了解到该符号是从.dll 导入的。链接时不执行任何验证。Delphi 生成一个引用符号/库的图像。如果该符号实际上并不在该库中，则可以在运行时找到它。
3. 在 Delphi 使用外部链接器的平台上，例如：当面向 iOSDevice64 平台时，通过外部指定的标识符<stringconstant>将传递给外部链接器。
Delphi 编译器传递给 <stringconstant> ld.exe。如果找不到库，则显示以下错误：Error: E2597 ld: file not found: <stringconstant>.

从.obj 导入函数（外部连接器）

使用外部链接器时，可以通过使用 external 指令指定对象文件来消除对 $L（或$LINK）编译器指令的使用。例如：

procedure FunctionName; cdecl; external object 'ObjectFile.o' name '_FunctionName';

从框架导入函数

您可以从外部框架导入例程。例如：

Function foo: Integer; external framework '<framework name>'

这仅适用于 iOS64、macOS64 ARM 和 macOS64。

修改导入函数名称

您可以导入与库中不同的例程名称。如果执行此操作，请在外部指令中指定原始名称：

external <stringConstant1> name <stringConstant2>;

where <stringConstant1> 提供库文件的名称，<stringConstant2>并且是例程的原始名称。

以下声明从 user32.dll（Windows API 的一部分）导入一个函数：

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

该函数的原始名称是 MessageBoxA，但它作为 MessageBox 导入。
在导入声明中，请确保与例程名称的确切拼写和大小写匹配。稍后，当您调用导入的例程时，名称不区分大小写。

通过 index 导入函数

您可以使用数字来标识要导入的例程，而不是名称：

external <stringConstant> index <integerConstant>;

其中 <integerConstant> 是导出表中例程的索引。

延迟导入

要将包含函数的库的加载推迟到实际需要该函数的那一刻，请将 delayed 指令附加到导入的函数：

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

延迟可确保包含导入函数的库不会在应用程序启动时加载，而是在首次调用函数时加载。有关本主题的详细信息，请参阅库和包 - 延迟加载主题。

指定库的依赖关系

如果包含目标例程的库依赖于其他库，请使用依赖项指令指定这些依赖项。

若要使用 dependency 指令，请附加 dependency 关键字，后跟逗号分隔的字符串列表。每个字符串必须包含作为目标外部库依赖项的库的名称：

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

以下声明表明 libmidas.a 依赖于标准 C++库：

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

重载

可以在同一作用域中声明多个具有相同名称的例程。这称为重载。重载例程必须使用 overload 指令声明，并且必须具有区分参数列表。例如，考虑以下声明：

```
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;
```

这些声明创建两个函数，都称为 Divide，它们采用不同类型的参数。调用 Division 时，编译器通过查看调用中传递的实际参数来确定要调用的函数。例如，Divide（6.0， 3.0） 调用第一个 Divide 函数，因为它的参数是实值的。

可以将类型与例程的任何声明中的参数类型不同，但与多个声明中的参数赋值兼容的参数传递给重载例程。当例程重载不同的整数类型或不同的实数类型时，这种情况最常发生 - 例如：

```
  procedure Store(X: Longint); overload;
  procedure Store(X: Shortint); overload;
```

在这些情况下，如果可以做到没有歧义，编译器将调用其参数类型为最小范围的例程，该范围可容纳调用中的实际参数。（请记住，实值常量表达式始终为扩展类型。）

重载例程必须通过它们采用的参数数或其参数类型来区分。因此，以下一对声明会导致编译错误：

```
function Cap(S: string): string; overload;
  ...
procedure Cap(var Str: string); overload;
```

但声明：

```
function Func(X: Real; Y: Integer): Real; overload;
  ...
function Func(X: Integer; Y: Real): Real; overload;
```

是合法的。

在转发或接口声明中声明重载例程时，定义声明必须重复例程的参数列表。

编译器可以区分在同一参数位置包含 AnsiString/PAnsiChar、UnicodeString/PChar 和 WideString/PWideChar 参数的重载函数。传递到这种重载情况下的字符串常量或文字被转换为本机字符串或字符类型，即 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;
```

变体也可以用作重载函数声明中的参数。变体被认为比任何简单类型都更通用。始终优先选择完全类型匹配而不是变体匹配。如果将变体传递到此类重载情况，并且该参数位置存在采用变体的重载，则认为它与 Variant 类型完全匹配。

这可能会对浮点类型造成一些轻微的副作用。浮子类型按大小匹配。如果传递给重载调用的浮点变量没有完全匹配项，但有变体参数可用，则该变量将接管任何较小的浮点类型。

例如：

```
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;
```

此示例调用 foo 的变体版本，而不是双精度版本，因为 1.2 常量隐式是扩展类型，而扩展不是双精度的完全匹配。扩展也不是 Variant 的完全匹配，但 Variant 被认为是更通用的类型（而 double 是比扩展更小的类型）。

```
foo(Double(1.2));
```

此类型转换不起作用。您应该改用类型化的常量：

```
const    d: double = 1.2;
   begin
      foo(d);
   end;
```

上面的代码工作正常，并调用双精度版本。

```
const    s: single = 1.2;
   begin
      foo(s);
   end;
```

上面的代码还调用了 foo 的双精度版本。单比变种更适合双倍。

声明一组重载例程时，避免浮点提升为变体的最佳方法是为每个浮点类型（单、双、扩展）声明重载函数的版本以及变体版本。

如果在重载例程中使用默认参数，请注意不要引入不明确的参数签名。

可以通过在调用例程时限定例程的名称来限制重载的潜在影响。例如，Unit1.MyProcedure（X， Y） 只能调用在 Unit1 中声明的例程;如果 Unit1 中没有例程与调用中的名称和参数列表匹配，则会产生错误。

本地声明

函数或过程的主体通常以例程的语句块中使用的局部变量的声明开头。这些声明还可以包括常量、类型和其他例程。本地标识符的作用域仅限于声明它的例程。

嵌套例程

函数和过程有时在其块的本地声明部分包含其他函数和过程。例如，以下名为 DoSomething 的过程声明包含一个嵌套过程。

```
procedure DoSomething(S: string);
  var
    X, Y: Integer;

    procedure NestedProc(S: string);
    begin
    ...
    end;

  begin
    ...
    NestedProc(S);
    ...
  end;
```

嵌套例程的作用域仅限于在其中声明它的过程或函数。在我们的示例中，只能在 DoSomething 中调用 NestedProc。

有关嵌套例程的真实示例，请查看 DateTimeToString 过程、ScanDate 函数和 SysUtils 单元中的其他例程。

参数

关于参数

大多数过程和函数标头都包含参数列表。例如，在标头中：

```
function Power(X: Real; Y: Integer): Real;
```

参数列表为 （X: Real; Y: Integer）。

参数列表是用分号分隔并括在括号中的参数声明序列。每个声明都是一系列以逗号分隔的参数名称，在大多数情况下后跟冒号和类型标识符，在某些情况下后跟 = 符号和默认值。参数名称必须是有效的标识符。任何声明前面都可以有 var、const 或 out。例子：

```
(X, Y: Real)(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

参数列表指定调用例程时必须传递给例程的参数的数量、顺序和类型。如果例程不采用任何参数，则省略其声明中的标识符列表和括号：

```
procedure UpdateRecords;
begin
  ...
end;
```

在过程或函数体中，参数名称（第一个示例中的 X 和 Y）可用作局部变量。不要在过程或函数体的本地声明部分中重新声明参数名称。

参数语义

参数按多种方式分类：

- ✓ 每个参数都分为值、变量、常量或 out。值参数是默认的;保留字 var、const 和 out 分别表示变量、常量和 out 参数。
- ✓ 值参数始终是类型化的，而常量、变量和 out 参数可以是类型化的，也可以是非类型化的。

特殊规则适用于数组参数。

包含文件的文件和结构化类型的实例只能作为变量 （var） 参数传递。

值和变量参数

大多数参数是值参数（默认）或变量（var）参数。值参数按值传递，而变量参数按引用传递。若要了解这意味着什么，请考虑以下函数：

```
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;
```

这些函数返回相同的结果，但只有第二个 - DoubleByRef 可以更改传递给它的变量的值。假设我们像这样调用函数：

```
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;
```

执行此代码后，传递给 DoubleByValue 的变量 I 具有我们最初分配给它的相同值。但是传递给 DoubleByRef 的变量 V 具有不同的值。

value 参数的作用类似于初始化为过程或函数调用中传递的值的局部变量。如果将变量作为值参数传递，则过程或函数将创建该变量的副本;对副本所做的更改对原始变量没有影响，并且在程序执行返回到调用方时会丢失。

另一方面，变量参数的作用类似于指针而不是副本。在程序执行返回到调用方并且参数名称本身超出范围后，对函数或过程主体中的参数所做的更改仍然存在。

即使在两个或多个 var 参数中传递相同的变量，也不会创建副本。以下示例对此进行了说明：

```
procedure AddOne(var X, Y: Integer);
begin
  X := X + 1;
  Y := Y + 1;
end;
var I: Integer;
begin
  I := 1;
  AddOne(I, I);
end;
```

执行此代码后，I 的值为 3。

如果例程的声明指定了 var 参数，则必须在调用例程时将可赋值表达式（即变量、类型化常量（处于 {$J+} 状态）、取消引用的指针、字段或索引变量传递给该例程。使用我们之前

的例子，DoubleByRef（7） 会产生一个错误，尽管 DoubleByValue（7） 是合法的。

在 var 参数中传递的索引和指针取消引用 - 例如，DoubleByRef（MyArray[I]） - 在执行例程之前被计算一次。

常量参数

常量 （const） 参数类似于局部常量或只读变量。常量参数类似于值参数，不同之处在于不能将值分配给过程或函数主体中的常量参数，也不能将一个作为 var 参数传递给另一个例程。（但是，当您将对象引用作为常量参数传递时，您仍然可以修改对象的属性。）

使用 const 允许编译器针对结构化和字符串类型参数优化代码。它还提供了一种保护措施，防止通过引用将参数无意中传递到另一个例程。

例如，这里是 SysUtils 单元中 CompareStr 函数的标头：

```
function CompareStr(const S1, S2: string): Integer;
```

由于 S1 和 S2 未在 CompareStR 的主体中修改，因此可以将它们声明为常量参数。

常量参数可以通过值或引用传递给函数，具体取决于所使用的特定编译器。若要强制编译器通过引用传递常量参数，可以将 [Ref] 修饰器与 const 关键字一起使用。

下面的示例演示如何在 const 关键字之前或之后指定 [Ref] 修饰器：

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

输出参数

输出参数 out 参数（如变量参数）通过引用传递。但是，使用 out 参数时，引用变量的初始值将被传递给它的例程丢弃。out 参数仅用于输出;也就是说，它告诉函数或过程在哪里存储输出，但不提供任何输入。

例如，考虑过程标题：

```
procedure GetInfo(out Info: SomeRecordType);
```

调用 GetInformation 时，必须向其传递一个类型为 SomeRecordType 的变量：

```
var MyRecord: SomeRecordType;
    ...
GetInfo(MyRecord);
```

但是您没有使用我的记录将任何数据传递给 GetInfo 过程;MyRecord 只是一个容器，您希望 GetInfo 在其中存储它生成的信息。对 GetInfo 的调用会立即释放 MyRecord 使用的内存，然后再将程序控制传递给过程。

输出参数经常用于分布式对象模型（如 COM）。此外，在将未初始化的变量传递给函数或过程时，应使用 out 参数。

无类型化参数

声明 var、const 和 out 参数时，可以省略类型规范。（必须键入值参数。）例如：

```
procedure TakeAnything(const C);
```

声明一个名为 TakeAnything 的过程，该过程接受任何类型的参数。调用此类例程时，不能向其传递数字或非类型化数字常量。

在过程或函数体中，非类型化参数与每种类型都不兼容。若要对非类型化参数进行操作，必须强制转换该参数。通常，编译器无法验证对非类型化参数的操作是否有效。

下面的示例在名为 Equal 的函数中使用非类型化参数，该函数比较任意两个变量的指定字节数：

```
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;
```

鉴于声明：

```
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;
```

您可以对 Equal 进行以下调用：

```
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
```

注意：FreeAndNil 不接受非类型化的 var 参数。
例如，要强制转换 Obj，应键入：
procedure FreeAndNil（var Obj： TObject）;

字符串参数

声明采用短字符串参数的例程时，不能在参数声明中包含长度说明符。也就是说，以下声明会导致编译错误：

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

但以下声明有效：

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

特殊标识符 OpenString 可用于声明采用不同长度的短字符串参数的例程：

```
procedure Check(S: OpenString);
```

当 {$H} 和 {$P+} 编译器指令都有效时，保留字 string 等效于参数声明中的 OpenString。
短字符串、OpenString、$H 和 $P 仅支持向后兼容。在新代码中，可以通过使用长字符串来避免这些注意事项。

数组参数

声明采用数组参数的例程时，不能在参数声明中包含索引类型说明符。也就是说，声明：

```
procedure Sort(A: array[1..10] of Integer)   // syntax error
```

导致编译错误。但：

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

有效。另一种方法是使用开放数组参数。
由于 Delphi 语言没有实现动态数组的值语义，因此例程中的“value”参数并不表示动态数组的完整副本。在此示例中：

```
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;
```

请注意，在例程 p 中对 Value[0] 的赋值将修改调用者的动态数组的内容，尽管 Value 是按值参数。如果需要动态数组的完整副本，请使用复制标准过程创建动态数组的值副本。

开放数组参数

开放数组参数允许将不同大小的数组传递给相同的过程或函数。要使用开放数组参数定义例程，请使用类型（而不是 array[X..Y] 的类型）在参数声明中。例如：

```
function Find(A: array of Char): Integer;
```

声明一个名为 Find 的函数，该函数采用任意大小的字符数组并返回一个整数。

注意：开放数组参数的语法类似于动态数组类型的语法，但它们的含义不同。前面的示例创建一个函数，该函数接受任何 Char 元素数组，包括（但不限于）动态数组。若要声明必须是动态数组的参数，需要指定类型标识符：

```
type TDynamicCharArray = array of Char;
function Find(A: TDynamicCharArray): Integer;
```

在例程的主体中，开放数组参数受以下规则的约束：

- ✓ 它们始终从零开始。第一个元素为 0，第二个元素为 1，依此类推。标准 Low 和 High 函数分别返回 0 和 Length - 1。SizeOf 函数返回传递给例程的实际数组的大小。
- ✓ 它们只能通过元素访问。不允许对整个开放数组参数进行赋值。
- ✓ 它们只能作为开放数组参数或非类型化 var 参数传递给其他过程和函数。它们不能传递给 SetLength。
- ✓ 您可以传递开放数组参数基类型的变量，而不是数组。它将被视为长度为 1 的数组。

将数组作为开放数组值参数传递时，编译器会在例程的堆栈帧中创建数组的本地副本。注意不要通过传递大型数组使堆栈溢出。

以下示例使用开放数组参数来定义一个 Clear 过程，该过程为实数数组中的每个元素赋值零，以及一个 Sum 函数，用于计算实数数组中元素的总和：

```
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;
```

调用使用开放数组参数的例程时，可以将开放数组构造函数传递给它们。

变体开放数组参数

变体开放数组参数允许您将不同类型表达式的数组传递给单个过程或函数。要使用变体开放数组参数定义例程，请将 const 数组指定为参数的类型。因此：

```
procedure DoSomething(A: array of const);
```

声明一个名为 DoSomething 的过程，该过程可以在变体数组上运行。

常量构造数组等效于 TVarRec 数组。System.TVarRec 在系统单元中声明，表示具有变体部分的记录，该变量部分可以保存整数、布尔值、字符、实数、字符串、指针、类、类引用、接口和变量类型的值。TVarRec 的 VType 字段指示数组中每个元素的类型。某些类型作为指针而不是值传递;特别是，字符串作为指针传递，并且必须类型转换为字符串。

下面的 Win32 示例在函数中使用变体开放数组参数，该参数创建传递给它的每个元素的字符串表示形式，并将结果连接成单个字符串。此函数中调用的字符串处理例程在 SysUtils 中定义：

```
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;
            vtWideChar: Result := Result + VWideChar;
            vtPWideChar: Result := Result + VPWideChar;
            vtAnsiString:    Result := Result + string(AnsiString(VAnsiString));
            vtUnicodeString:    Result := Result + string(UnicodeString(VUnicodeString));
            vtCurrency:       Result := Result + CurrToStr(VCurrency^);
            vtVariant:        Result := Result + string(VVariant^);
            vtWideString:    Result := Result + string(WideString(VWideString));
            vtInt64:          Result := Result + IntToStr(VInt64^);
  end;
end;
```

我们可以使用开放数组构造函数调用此函数。例如：

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

返回字符串“test100 T3.14159TForm”。

默认参数

可以在过程或函数标题中指定默认参数值。仅允许类型化的常量和值参数使用默认值。若要提供默认值，请使用 = 符号结束参数声明，后跟与参数类型赋值兼容的常量表达式。

例如，给定声明：

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

以下过程调用是等效的。

```
FillArray(MyArray);
FillArray(MyArray, 0);
```

多参数声明不能指定默认值。因此，虽然以下声明是合法的：

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

但以下声明不合法：

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

具有默认值的参数必须出现在参数列表的末尾。也就是说，第一个声明的默认值后面的所有参数也必须具有默认值。因此，以下声明是非法的：

```
procedure MyProcedure(I: Integer = 1; S: string);    // syntax error
```

在过程类型中指定的默认值将覆盖实际例程中指定的默认值。因此，鉴于声明：

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
   F: TResizer;
   N: Real;
```

执行语句：

```
F := Resizer;
F(N);
```

导致值 （N， 1.0） 被传递到调整器。

默认参数仅限于可由常量表达式指定的值。因此，动态数组、过程、类、类引用或接口类型的参数除了 nil 之外不能有默认值。记录、变体、文件、静态数组或对象类型的参数根本不能具有默认值。

默认参数和重载函数

如果在重载例程中使用默认参数值，请避免使用不明确的参数签名。例如，请考虑以下事项：

```
procedure Confused(I: Integer); overload;
    ...
procedure Confused(I: Integer; J: Integer = 0); overload;
    ...
Confused(X);     //    Which procedure is called?
```

事实上，这两个过程都没有调用。此代码生成编译错误。

前向声明和接口声明中的默认参数

如果例程具有前向声明或出现在单元的接口部分中，则必须在前向或接口声明中指定默认参数值（如果有）。在这种情况下，可以从定义（实现）声明中省略默认值;但是，如果定义声明包含默认值，则它们必须与转发或接口声明中的默认值完全匹配。

调用

程序控制和参数

调用过程或函数时，程序控制从调用的点传递到例程的主体。可以使用例程的声明名称（带或不带限定符）或使用指向例程的过程变量进行调用。在任一情况下，如果使用参数声明例程，则对它的调用必须将按顺序和类型对应的参数传递给例程的参数列表。传递给例程的参数称为实际参数，而例程声明中的参数称为形式参数。

调用例程时，请记住：

- ✓ 用于传递类型化的 const 和值参数的表达式必须与相应的形式参数赋值兼容。
- ✓ 用于传递 var 和 out 参数的表达式必须与相应的形式参数相同，除非形式参数是非类型化的。
- ✓ 只有可赋值的表达式才能用于传递 var 和 out 参数。
- ✓ 如果例程的形式参数是非类型化的，则数字和带有数值的普通常量不能用作实际参数。

调用使用默认参数值的例程时，第一个接受的默认值之后的所有实际参数也必须使用默认值;调用形式为 SomeFunction(,,X) 是不合法的。

将所有参数（仅默认参数）传递给例程时，可以省略括号。例如，给定过程：

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

以下调用是等效的：

```
DoSomething();
DoSomething;
```

开放数组构造函数

开放数组构造函数允许您直接在函数和过程调用中构造数组。它们只能作为开放数组参数或变体开放数组参数传递。

与集合构造函数一样，开放数组构造函数是用逗号分隔并括在括号中的表达式序列。

例如，给定声明：

```
var I, J: Integer;
```

```
procedure Add(A: array of Integer);
```

您可以使用以下语句调用 Add 过程：

```
Add([5, 7, I, I + J]);
```

这相当于：

```
var Temp: array[0..3] of Integer;
  // …
  Temp[0] := 5;
  Temp[1] := 7;
  Temp[2] := I;
  Temp[3] := I + J;
  Add(Temp);
```

开放数组构造函数只能作为值或常量参数传递。构造函数中的表达式必须与数组参数的基类型赋值兼容。对于变体开放数组参数，表达式可以是不同的类型。

使用内联指令

使用内联指令 Delphi 编译器允许使用内联指令标记函数和过程以提高性能。如果函数或过程满足特定条件，编译器将直接插入代码，而不是生成调用。内联是一种性能优化，可以加快代码速度，但会占用空间。内联始终会导致编译器生成更大的二进制文件。与其他指令一样，内联指令用于函数和过程声明和定义。

```
procedure MyProc(x:Integer); inline;
begin
      // …
end;
function MyFunc(y:Char) : String; inline;
begin
  // …
end;
```

内联指令是对编译器的建议。不能保证编译器将内联特定例程，因为在许多情况下无法内联。以下列表显示了内联发生或不发生内联的条件：

- ✓ 内联不会发生在任何形式的后期绑定方法上。这包括虚拟、动态和消息方法。
- ✓ 包含汇编代码的例程将不会内联。
- ✓ 构造函数和析构函数不会内联。
- ✓ 主程序块、单元初始化和单元终结块不能内联。
- ✓ 使用前未定义的例程不能内联。
- ✓ 不能内联采用开放数组参数的例程。

- ✓ 代码可以在包中内联，但是，内联永远不会跨包边界发生。
- ✓ 在循环依赖的单元之间不执行内联。这包括间接循环依赖关系，例如，单元 A 使用单元 B，单元 B 使用单元 C，而单元 C 又使用单元 A。在此示例中，编译单元 A 时，单元 B 或单元 C 中的代码不会内联到单元 A 中。
- ✓ 当单元处于循环依赖关系中时，编译器可以内联代码，只要要内联的代码来自循环关系之外的单元即可。在上面的示例中，如果单元 A 也使用单元 D，则单元 D 中的代码可以内联在 A 中，因为它不参与循环依赖关系。
- ✓ 如果在接口部分中定义了例程，并且它访问在实现部分中定义的符号，则无法内联该例程。
- ✓ 如果标有内联的例程使用来自其他单位的外部符号，则必须在 uses 语句中列出所有这些单位，否则无法内联该例程。
- ✓ 在一个单元中，应在调用内联函数之前定义该函数的主体。否则，编译器在到达调用站点时不知道的函数主体无法内联扩展。

如果修改内联例程的实现，将导致重新编译使用该函数的所有单元。这与传统的重建规则不同，在传统的重建规则中，重建仅由单元界面部分的更改触发。

{$INLINE} 编译器指令使你能够更精细地控制内联。{$INLINE} 指令可以在例程的内联定义的站点以及调用站点使用。以下是可能的值及其含义：

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.

匿名方法

顾名思义，匿名方法是没有与之关联的名称的过程或函数。匿名方法将代码块视为可以分配给变量或用作方法参数的实体。此外，匿名方法可以引用变量并将值绑定到定义该方法的上下文中的变量。匿名方法可以通过简单的语法定义和使用。它们类似于其他语言中定义的闭包结构。

语法

匿名方法的定义类似于常规过程或函数，但没有名称。
例如，此函数返回一个定义为匿名方法的函数：

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

```
  begin
    Result := x + y;
  end; { end anonymous method }
end;
```

函数 MakeAdder 返回一个它声明的没有名称的函数：一个匿名方法。

请注意，MakeAdder 返回一个类型为 TFuncOfInt 的值。匿名方法类型声明为对方法的引用：

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

此声明指示匿名方法：

- ✓ 是一个函数
- ✓ 采用一个整数参数
- ✓ 返回一个整数值。

通常，为过程或函数声明匿名函数类型：

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

其中(parameterlist)是可选的。

下面是几个类型的示例：

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

匿名方法声明为没有名称的过程或函数：

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

其中 (parameters)是可选的。

用法

匿名方法通常分配给某些内容，如以下示例所示：

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

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

匿名方法也可以由函数返回，或者在调用方法时作为参数的值传递。例如，使用上面定义的匿名方法变量 myFunc：

```
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;)
```

方法引用也可以分配给方法以及匿名方法。例如：

```
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;
```

但是，反之则不然：不能将匿名方法分配给常规方法指针。方法引用是托管类型，但方法指针是非托管类型。因此，出于类型安全原因，不支持将方法引用分配给方法指针。例如，事件是方法指针值属性，因此不能对事件使用匿名方法。有关此限制的详细信息，请参阅变量绑定部分。

变量绑定

匿名方法的一个关键特征是，它们可以引用在定义它们的位置可见的变量。此外，这些变量可以绑定到值，并使用对匿名方法的引用进行包装。这将捕获状态并延长变量的生存期。

绑定示例

再次考虑上面定义的函数：

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

我们可以创建这个函数的实例来绑定一个变量值：

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

变量 adder 包含一个匿名方法，该方法将值 20 绑定到匿名方法的代码块中引用的变量

y。即使值超出范围，此绑定也会保留。

事件匿名方法

使用方法引用的动机是具有可以包含绑定变量（也称为闭包值）的类型。由于闭包在其定义环境中关闭，包括在定义点引用的任何局部变量，因此它们具有必须释放的状态。方法引用是托管类型（它们是引用计数的），因此它们可以跟踪此状态并在必要时释放它。如果方法引用或闭包可以自由地分配给方法指针（例如事件），那么很容易创建带有悬空指针或内存泄漏的格式不正确的程序。

Delphi 事件是属性的约定。事件和属性之间没有区别，除了类型。如果属性是方法指针类型，则它是一个事件。

如果属性是方法引用类型，则在逻辑上也应将其视为事件。但是，IDE 不会将其视为事件。这对于作为组件和自定义控件安装到 IDE 中的类非常重要。

因此，若要在组件或自定义控件上具有可以使用方法引用或闭包值分配给的事件，该属性必须是方法引用类型。但是，这很不方便，因为 IDE 不会将其识别为事件。

下面是使用具有方法引用类型的属性的示例，以便它可以作为事件运行：

```
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;
```

变量绑定机制

为了避免产生内存泄漏，更详细地了解变量绑定过程很有用。

在过程、函数或方法（以下简称“例程”）开始时定义的局部变量通常仅在该例程处于活动

状态时才存在。匿名方法可以延长这些变量的生存期。

如果匿名方法在其主体中引用外部局部变量，则该变量将被“捕获”。捕获意味着延长变量的生存期，以便它与匿名方法值一样长，而不是随着其声明例程而死亡。请注意，变量捕获捕获的是变量，而不是值。如果变量的值在通过构造匿名方法捕获后发生变化，则匿名方法捕获的变量的值也会更改，因为它们是具有相同存储的同一变量。捕获的变量存储在堆上，而不是堆栈上。

匿名方法值属于方法引用类型，并且是引用计数的。当对给定匿名方法值的最后一个方法引用超出范围，或者被清除（初始化为 nil）或最终确定时，它捕获的变量最终会超出范围。

在多个匿名方法捕获同一局部变量的情况下，这种情况更为复杂。要了解这在所有情况下是如何工作的，有必要更精确地了解实现的机制。

每当捕获局部变量时，都会将其添加到与其声明例程关联的“帧对象”中。例程中声明的每个匿名方法都将转换为与其包含例程关联的帧对象上的方法。最后，由于正在构造的匿名方法值或正在捕获的变量而创建的任何帧对象都将通过另一个引用链接到其父帧 - 如果存在任何此类帧，并且如果需要访问捕获的外部变量。从一个帧对象到其父对象的这些链接也被引用计数。在嵌套的本地例程中声明的匿名方法从其父例程中捕获变量，使父帧对象保持活动状态，直到它本身超出范围。

例如，请考虑以下情况：

```
type
  TProc = reference to procedure;procedure Call(proc: TProc);
// ...
procedure Use(x: Integer);
// ...
procedure L1; // frame F1var
  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;

每个例程和匿名方法都使用帧标识符进行批注，以便更轻松地识别哪个帧对象链接到哪个帧对象：

- ✓ v1 是 F1 中的一个变量
- ✓ v2 是 F1_2 中的变量（由 F1_2_1 捕获）
- ✓ 匿名 F1_1_1 方法是 F1_1 中的一种方法
- ✓ F1_1 链接到 F1（F1_1_1 使用 v1）
- ✓ F1_2 的匿名方法是 F1 中的一种方法
- ✓ 匿名 F1_2_1 方法是 F1_2 中的一种方法

帧 F1_2_1 和 F1_1_1 不需要帧对象，因为它们既不声明匿名方法，也不具有捕获的变量。它们也不在嵌套匿名方法和外部捕获变量之间的任何父级路径上。（它们在堆栈上存储了隐式帧。

仅给定对匿名方法 F1_2_1 的引用，变量 v1 和 v2 保持活动状态。相反，如果唯一比 F1 调用更久的引用是 F1_1_1，则只有变量 v1 保持活动状态。

可以在方法引用/帧链接链中创建导致内存泄漏的循环。例如，将匿名方法直接或间接存储在匿名方法本身捕获的变量中会产生一个循环，从而导致内存泄漏。

实用技巧

匿名方法提供的不仅仅是指向可调用内容的简单指针。它们具有以下几个优点：

- ✓ 绑定变量值
- ✓ 定义和使用方法的简单方法
- ✓ 使用代码轻松参数化

变量绑定

匿名方法向定义它们的环境提供代码块以及变量绑定，即使该环境不在范围内也是如此。指向函数或过程的指针无法执行此操作。

例如，语句 adder ：= MakeAdder（20）;从上面的代码示例中生成一个变量 Adder，该 Adder 将变量与值 20 的绑定封装在一起。

实现这种结构的其他一些语言将它们称为闭包。从历史上看，这个想法是计算像 adder 这样的表达式 ：= MakeAdder（20）;产生了一个闭合。它表示一个对象，该对象包含对函数中引用并在函数外部定义的所有变量的绑定的引用，从而通过捕获变量的值来关闭它。

易用性

下面的示例演示一个典型的类定义，用于定义一些简单的方法，然后调用它们：

```
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.
```

将此方法与使用匿名方法定义和调用的相同方法进行对比：

```
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.
```

请注意，使用匿名方法的代码是多么简单和简短。如果要显式简单地定义这些方法并立即使用它们，而无需创建可能永远不会在其他任何地方使用的类的开销和工作量，则这是理想的选择。生成的代码更易于理解。

代码参数

匿名方法可以更轻松地编写由代码（而不仅仅是值）参数化的函数和结构。

多线程是匿名方法的一个很好的应用程序。如果要并行执行某些代码，则可能有一个如下所示的 parallel-for 函数：

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

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

ParallelFor 过程在不同的线程上循环访问过程。假设使用线程或线程池正确有效地实现此过程，则可以轻松地使用它来利用多个处理器：

```
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;
```

与此形成对比的是，在没有匿名方法的情况下，它需要完成的方式：可能是一个带有虚拟抽象方法的“任务”类，具有 ExpensiveComputing 的具体后代，然后将所有任务添加到队列中 - 几乎不那么自然或集成。

在这里，“并行”算法是由代码参数化的抽象。过去，实现此模式的常用方法是使用具有一个或多个抽象方法的虚拟基类;考虑 TThread 类及其抽象的 Execute 方法。但是，匿名方法使这种模式（使用代码对算法和数据结构进行参数化）变得更加容易。

6. 类和对象

类和对象

类类型

类或类类型定义由字段、方法和属性组成的结构。类类型的实例称为对象。类的字段、方法和属性称为其组件或成员。

字段本质上是作为对象一部分的变量。与记录的字段一样，类的字段表示存在于类的每个实例中的数据项。

方法是与类关联的过程或函数。大多数方法对对象（即类的实例）进行操作。某些方法（称为类方法）对类类型本身进行操作。

属性是与对象关联的数据的接口（通常存储在字段中）。属性具有访问说明符，这些说明符确定如何读取和修改其数据。从对象本身之外的程序的其他部分，属性在大多数方面都像字段一样出现。

对象是动态分配的内存块，其结构由其类类型确定。每个对象都有类中定义的每个字段的唯一副本，但类的所有实例共享相同的方法。对象由称为构造函数和析构函数的特殊方法创建和销毁。

类类型的变量实际上是引用对象的指针。因此，多个变量可以引用同一个对象。与其他指针一样，类类型变量可以保存值 nil。但是，您不必显式取消引用类类型变量来访问它指向的对象。例如，SomeObject.Size ：= 100 将值 100 分配给 SomeObject 引用的对象的 Size 属性;你不会把它写成 SomeObject^。大小 ：= 100。

必须先声明类类型并为其指定名称，然后才能实例化。（不能在变量声明中定义类类型。仅在程序或单元的最外围声明类，而不在过程或函数声明中声明类。

类类型声明具有以下形式：

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

类类型声明的必需元素：

- ✓ 类名是任何有效的标识符。
- ✓ memberList 声明类的成员：字段、方法和属性。

类类型声明的可选元素：

- ✓ abstract 整个类可以声明为抽象，即使它不包含任何抽象的虚拟方法。
- ✓ Sealed 密封类不能通过继承进行扩展。
- ✓ ancestorClass 新类直接从预定义的 System.TObject 类继承，可以省略 （ancestorClass）。如果包含 （ancestorClass），并且 memberList 为空，则可以省略 end。
- ✓ nestedTypeDeclaration 嵌套类型提供了一种将概念上相关的类型保持在一起并避免名称冲突的方法。
- ✓ nestedConstDeclaration 嵌套常量提供了一种将概念上相关的类型保持在一起并避免名称冲突的方法。

类不能既抽象又密封。[抽象 | 密封] 语法（[ ] 括号和它们之间的 | 管道）用于指定只能使用一个可选的密封或抽象关键字。只有密封或抽象的关键字才有意义。括号和管道符号应予删除。

注意：为了向后兼容，Delphi 允许实例化声明为 abstract 的类，但不应再使用此功能。

方法在类声明中显示为函数或过程标题，没有主体。为每个方法定义声明发生在程序的其他位置。

例如，下面是来自类单元的 TMemoryStream 类的声明：

```
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 是从 Classes.TCustomMemoryStream 衍生而来的，继承了它的大部分成员。但它定义或重新定义了几种方法和属性，包括其析构函数方法 Destroy。它的构造函数 Create 是继承的，无需更改即可从 System.TObject 继承，因此不会重新声明。每个成员都声明为 price、protected 或 public（此类没有 published 的成员）。这些术语稍后解释。

根据此声明，您可以创建 TMemoryStream 的实例，如下所示：

```
var
  stream: TMemoryStream;
begin
  stream := TMemoryStream.Create;
```

继承和作用域

声明类时，可以指定其直接祖先。例如：

```
  type TSomeControl = class(TControl);
```

声明一个名为 TSomeControl 的类，该类源自 Vcl.Controls.TControl。类类型会自动从其直接祖先继承所有成员。每个类都可以声明新成员并可以重新定义继承的成员，但类不能删除祖先中定义的成员。因此，TSomeControl 包含 Vcl.Controls.TControl 和每个 Vcl.Controls.TControl 祖先中定义的所有成员。

成员标识符的作用域从声明成员的点开始，一直到类声明的末尾，并扩展到类的所有后代以及类及其后代中定义的所有方法的块。

TObject and TClass

在 System 单元中声明的 System.TObject 类是所有其他类的最终祖先。System.TObject 只定义了少数几个方法，包括基本的构造函数和析构函数。除了 System.TObject 之外，System 单元还声明了类引用类型 System.TClass：

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

如果类类型的声明未指定祖先，则该类直接从 System.TObject 继承。因此：

```
type TMyClass = class
       ...
     end;
```

相当于：

```
type TMyClass = class(TObject)
       ...
     end;
```

为了便于阅读，建议使用后一种形式。

类类型的兼容性

类类型与其祖先的赋值兼容。因此，类类型的变量可以引用任何后代类型的实例。例如，给定声明：

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure);
  TSquare = class(TRectangle);
var
 Fig: TFigure;
```

变量 Fig 可以分配类型为 TFigure、TRectangle 和 TSquare 的值。

对象类型

Delphi 编译器允许使用类类型的替代语法。您可以使用以下语法声明对象类型：

```
type objectTypeName = object (ancestorObjectType)
        memberList
    end;
```

其中 objectTypeName 是任何有效的标识符，（ancestorObjectType）是可选的，memberList 声明字段、方法和属性。如果省略 （ancestorObjectType），则新类型没有祖先。对象类型不能具有已发布的成员。

由于对象类型不是从 System.TObject 派生的，因此它们不提供内置构造函数、析构函数或其他方法。可以使用 New 过程创建对象类型的实例，并使用 Dispose 过程销毁它们，也可以简单地声明对象类型的变量，就像声明记录一样。

仅支持向后兼容的对象类型。不建议使用它们。

成员可见性

类的每个成员都有一个称为可见性的属性，该属性由保留字之一表示 private、protected、public、published 或 automated。例如

```
  published property Color: TColor read GetColor write SetColor;
```

声明名为 Color 的已发布属性。可见性确定访问成员的位置和方式，私有表示最低可访问性，受保护表示中等级别的可访问性，公共、已发布和自动表示最大可访问性。

如果成员的声明显示时没有自己的可见性说明符，则该成员的可见性与其前面的声明相同。默认情况下，将发布类声明开头不具有指定可见性的成员，前提是该类是在 {$M+} 状态下编译的，或者派生自以 {$M+} 状态编译的类;否则，此类成员是公开的。

为了便于阅读，最好按可见性组织类声明，将所有私有成员放在一起，然后是所有受保护的成员，依此类推。这样，每个可见性保留字最多出现一次，并标志着声明的新“部分”的开始。所以一个典型的类声明应该是这样的：

```
type
  TMyClass = class(TControl)
    private
      { private declarations here }
    protected
      { protected declarations here }
    public
      { public declarations here }
    published
      { published declarations here }
  end;
```

可以通过重新声明属性来增加属性在后代类中的可见性，但不能降低其可见性。例如，受保护的属性可以在后代中公开，但不能设为私有。此外，已发布的属性不能在后代类中公开。有关更多信息，请参见属性重写和重新声明。

Private, Protected, and Public 成员

私有成员在声明其类的单元或程序之外是不可见的。换句话说，不能从另一个模块调用私有方法，也不能从另一个模块读取或写入私有字段或属性。通过将相关的类声明放在同一个模块中，可以向每个类授予对另一个类的私有成员的访问权限，而不会使这些成员更广泛地访问。要使成员仅在其类中可见，需要将其声明为严格私有。

受保护成员在声明其类的模块中的任何位置以及从任何后代类中可见，而不考虑出现后代类的模块。可以从属于从声明受保护成员的类派生的类的任何方法的定义中调用受保护的方法，并读取或写入受保护的字段或属性。仅用于派生类实现的成员通常受到保护。

公共成员在可以引用其类的任何地方都可见。

严格私有

除了私有和受保护的可见性说明符外，Delphi 编译器还支持具有更大访问约束的其他可见性设置。这些设置是严格的专用和严格的受保护可见性。

具有严格私有可见性的类成员只能在声明它们的类中访问。它们对在同一单元中声明的过程或函数不可见。具有严格受保护可见性的类成员在声明它们的类中可见，并且在任何后代类中可见，无论在何处声明。此外，当实例成员（声明没有类或类 var 关键字的成员）被声明为严格私有或严格保护时，它们在出现它们的类的实例之外无法访问。类的实例无法访问同一类的其他实例中的严格私有或严格受保护的实例成员。

注意：严格一词在类声明的上下文中被视为指令。在类声明中，不能声明名为“strict”的成员，但可以在类声明之外使用。

Published 成员

已发布成员的可见性与公共成员相同。区别在于为已发布成员生成运行时类型信息（RTTI）。RTTI 允许应用程序动态查询对象的字段和属性并定位其方法。RTTI 用于在保存和加载表单文件时访问属性的值，在对象检查器中显示属性，以及将特定方法（称为事件处理程序）与特定属性（称为事件）相关联。

已发布的属性仅限于某些数据类型。可以发布序号、字符串、类、接口、变体和方法指针类型。设置类型也是如此，前提是基类型的上限和下限具有从 0 到 31 的序号值。（换句话说，集合必须适合字节、单词或双单词。除了 Real48 之外的任何真实类型都可以发布。无法发布数组类型的属性（不同于下面讨论的数组属性）。

某些属性虽然可发布，但流式处理系统并不完全支持。其中包括记录类型的属性、所有可发布类型的数组属性以及包含匿名值的枚举类型的属性。如果发布此类属性，则对象检查器将无法正确显示该属性，并且在将对象流式传输到磁盘时也不会保留该属性的值。

所有方法都是可发布的，但类不能发布两个或多个具有相同名称的重载方法。仅当字段属于类或接口类型时，才能发布字段。

类不能具有已发布成员，除非该类在 {$M+} 状态下编译或从以 {$M+} 状态编译的类派生。大多数具有已发布成员的类都派生自 Classes.TPersistent，该类在 {$M+} 状态下编译，因此很少需要使用 $M 指令。

注：类的已发布节或已发布成员使用的类型中不允许包含 Unicode 字符的标识符。

Automated 成员（win32)

自动化成员具有与公共成员相同的可见性。区别在于，自动化类型信息（自动化服务器需要）是为自动化成员生成的。自动成员通常只出现在 Win32 类中。维护自动保留字以实现向后兼容性。ComObj 单元中的 TAutoObject 类不使用 automated。

以下限制适用于声明为自动化的方法和属性：

- ✓ 所有属性的类型、数组属性参数、方法参数和函数结果必须是可自动化的。可自动化的类型包括字节、货币、实数、双精度、隆整数、单、小、安西字符串、宽字符串、TDateTime、变体、OleVariant 、WordBool 和所有接口类型。
- ✓ 方法声明必须使用默认的寄存器调用约定。它们可以是虚拟的，但不是动态的。
- ✓ 属性声明可以包含访问说明符（读取和写入），但不允许使用其他说明符（索引、存储、默认和无默认值）。访问说明符必须列出使用默认寄存器调用约定的方法标识符;不允许使用字段标识符。
- ✓ 属性 声明必须指定类型。不允许属性覆盖。

自动方法或属性的声明可以包含 dispid 指令。在 dispid 指令中指定已使用的 ID 会导致错误。

在 Win32 平台上，此指令后必须跟一个整数常量，该常量指定成员的自动化调度 ID。否则，编译器会自动为成员分配一个调度 ID，该 ID 比类中的任何方法或属性及其祖先使用的最大调度 ID 大一个。有关自动化（仅在 Win32 上）的详细信息，请参阅自动化对象。

前向声明和相互依赖的类

如果类类型的声明以单词 class 和分号结尾，也就是说，如果它具有:

```
type    className = class;
```

如果单词 class 后没有列出祖先或类成员，则它是一个前向声明。前向声明必须由同一类型声明节中同一类的定义声明解析。换句话说，在前向声明及其定义声明之间，除了其他类型的声明之外，什么都不会发生。

前向声明允许相互依赖的类。例如：

```
type
   TFigure = class;      // forward declaration
   TDrawing = class
      Figure: TFigure;
      // ...
   end;

   TFigure = class      // defining declaration
      Drawing: TDrawing;
      // ...
   end;
```

不要将前向声明与派生自 System.TObject 而不声明任何类成员的类型的完整声明混淆。

```
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
```

字段

字段概述

字段类似于属于对象的变量。字段可以是任何类型，包括类类型。（也就是说，字段可以保存对象引用。字段通常是私有的。

要定义类的字段成员，只需像声明变量一样声明该字段。例如，下面的声明创建一个名为 TNumber 的类，除了从 System.TObject 继承的方法之外，该类的唯一成员是一个名为 Int 的整数字段：

```
type
```

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

var 关键字是可选的。但是，如果未使用它，则所有字段声明都必须出现在任何属性或方法声明之前。在任何属性或方法声明之后，var 可用于引入任何其他字段声明。

字段是静态绑定的;也就是说，对它们的引用在编译时是固定的。若要了解这意味着什么，请考虑以下代码：

```
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;
```

尽管 MyObject 拥有 TDescendant 的实例，但它被声明为 TAncestor。因此，编译器将 MyObject.Value 解释为引用在 TAncestor 中声明的（整数）字段。但是，这两个字段都存在于 TDescendant 对象中;继承的值由新值隐藏，可以通过类型转换访问。

常量和类型化常量的声明可以出现在全局范围的类和非匿名记录中。常量和类型化常量也可以出现在嵌套类型定义中。仅当类在过程本地定义时，常量和类型化常量才能出现在类定义中（即，它们不能出现在过程内定义的记录中）。

类字段

类字段是类中的数据字段，可以在没有对象引用的情况下访问（与上面讨论的普通“实例字段”不同）。存储在类字段中的数据由类的所有实例共享，并且可以通过引用类或表示类实例的变量来访问。

可以使用 static var 块声明在类声明中引入类字段块。在 static var 之后声明的所有字段都具有静态存储属性。static var 块由以下项终止：

1. 另一个 static var 或 var 声明
2. 过程或函数（即方法）声明（包括类过程和类函数）
3. 属性声明（包括类属性）
4. 构造函数或析构函数声明
5. 可见性范围说明符（public, private, protected, published, strict private, and strict protected）

例如：

```
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;
```

可以使用代码访问类字段“红色”、“绿色”和“蓝色”：

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

类字段也可以通过类的实例进行访问。具有以下声明：

```
var
  myObject: TMyClass;
```

此代码与上面对红色、绿色和蓝色的赋值具有相同的效果：

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

方法

方法是与类关联的过程或函数。对方法的调用指定该方法应对其进行操作的对象（或者，如果是类方法，则指定类）。例如，SomeObject.Free 在 SomeObject 中调用 Free 方法。

方法概述

在类声明中，方法显示为过程和函数标题，其工作方式类似于前向声明。在类声明之后的某个地方，但在同一模块中，每个方法都必须由定义声明实现。例如，假设 TMyClass 的声明包含一个名为 DoSomething 的方法：

```
type
    TMyClass = class(TObject)
        ...
        procedure DoSomething;
        ...
    end;
```

DoSomething 的定义声明必须在模块的后面出现：

```
procedure TMyClass.DoSomething;
begin
      ...
end;
```

虽然类可以在单元的接口或实现部分中声明，但定义类方法的声明必须在实现部分中。

在定义声明的标题中，方法名称始终使用它所属的类的名称进行限定。标题可以重复类声明中的参数列表;如果是这样，则参数的顺序、类型和名称必须完全匹配，如果方法是函数，则返回值也必须匹配。

方法声明可以包含不与其他函数或过程一起使用的特殊指令。指令应仅出现在类声明中，而不出现在定义声明中，并且应始终按以下顺序列出：

reintroduce; overload; binding; calling convention; abstract; warning

其中：

- ✓ binding 是 virtual, dynamic, or override;
- ✓ calling convention 是 REGISTER、PASCAL、CDECL、STDCALL 或 SafeCall;
- ✓ warning 为 platform, deprecated, or library。有关这些警告（提示）指令的详细信息，请参阅《提示指令》。

所有 Delphi 指令都列在《指令》中。

Inherited

继承的保留字在实现多态行为中起着特殊作用。它可以出现在方法定义中，后面有或没有标识符。

如果继承后跟成员名称，则表示对属性或字段的常规方法调用或引用，但对引用成员的搜

索以封闭方法类的直接祖先开头。例如，当：

```
inherited Create(...);
```

发生在定义中的方法，它调用继承的 Create。

当继承的后面没有标识符时，它引用与封闭方法同名的继承方法，或者，如果封闭方法是消息处理程序，则引用同一消息的继承消息处理程序。在这种情况下，继承不采用显式参数，而是将调用封闭方法时使用的相同参数传递给继承的方法。例如：

```
inherited;
```

在构造函数的实现中经常发生。它使用传递给后代的相同参数调用继承的构造函数。

Self

在方法的实现中，标识符 Self 引用在其中调用该方法的对象。例如，下面是 Classes 单元中 TCollection Add 方法的实现：

```
function TCollection.Add: TCollectionItem;
  begin
       Result := FItemClass.Create(Self);
  end;
```

Add 方法调用 FItemClass 字段引用的类中的 Create 方法，该字段始终是 TCollectionItem 后代。TCollectionItem.Create 采用 TCollection 类型的单个参数，因此 Add 会向其传递调用 Add 的 TCollection 实例对象。以下代码对此进行了说明：

```
var MyCollection: TCollection;
       ...
       MyCollection.Add      // MyCollection is passed to the
                                 // TCollectionItem.Create method
```

Self 是有用的，原因有很多。例如，在类类型中声明的成员标识符可能会在类方法之一的块中重新声明。在这种情况下，您可以访问原始成员标识符作为 Self.Identifier。

有关类方法中的 Self 的信息，请参阅类引用中的《类运算符》。

方法绑定

方法绑定可以是静态的（默认）、虚拟的或动态的。虚拟和动态方法可以被覆盖，它们可以是抽象的。当一个类类型的变量包含后代类类型的值时，这些指定将发挥作用。它们确定在调用方法时激活哪个实现。

静态方法

默认情况下，方法是静态的。调用静态方法时，方法调用中使用的类或对象变量的声明（编译时）类型确定要激活的实现。在下面的示例中，Draw 方法是静态的：

```
type
    TFigure = class
      procedure Draw;
    end;

    TRectangle = class(TFigure)
      procedure Draw;
    end;
```

给定这些声明，下面的代码说明了调用静态方法的效果。在对 Figure.Draw 的第二次调用中，Figure 变量引用了类 TRectangle 的对象，但该调用调用了 TFigure 中 Draw 的实现，因为 Figure 变量的声明类型是 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;
```

虚拟和动态方法

若要使方法成为虚拟或动态方法，请在其声明中包含 virtual 或 dynamic 指令。与静态方法不同，虚拟和动态方法可以在后代类中重写。调用重写的方法时，方法调用中使用的类或对象的实际（运行时）类型（而不是变量的声明类型）确定要激活的实现。

若要重写方法，请使用 override 指令重新声明该方法。重写声明必须在参数的顺序和类型以及结果类型（如果有）上与祖先声明匹配。

在下面的示例中，在两个后代类中重写了在 TFigure 中声明的 Draw 方法：

```
type
  TFigure = class
    procedure Draw; virtual;
  end;

  TRectangle = class(TFigure)
    procedure Draw; override;
  end;

  TEllipse = class(TFigure)
    procedure Draw; override;
  end;
```

给定这些声明，下面的代码说明了通过实际类型在运行时变化的变量调用虚拟方法的效果：

```
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;
```

只能重写虚拟和动态方法。但是，所有方法都可以重载;请参阅《重载方法》。

最终方法

Delphi 编译器还支持最终虚拟和动态方法的概念。最终方法的声明具有以下形式：

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

这里使用虚拟|动态语法（两个关键字和它们之间的 | 管道）来指定应使用一个且仅使用一个虚拟或动态关键字。有意义的只是虚拟或动态关键字;应删除管道符号本身。

当关键字 final 应用于虚拟或动态方法时，任何后代类都不能重写该方法。使用 final 关键字是一项重要的设计决策，可帮助记录如何使用类。它还可以为编译器提供提示，使其能够

优化生成的代码。

注意：虚拟关键字或动态关键字必须在最终关键字之前写入。
例

```
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;
```

虚拟与动态

在 Delphi for Win32 中，虚拟方法和动态方法在语义上是等效的。但是，它们在运行时实现方法调用调度时有所不同：虚拟方法针对速度进行优化，而动态方法针对代码大小进行优化。

通常，虚拟方法是实现多态行为的最有效方法。当基类声明许多可重写的方法时，动态方法非常有用，这些方法由应用程序中的许多后代类继承，但只是偶尔重写。

注意：仅当有明显、可观察到的好处时才使用动态方法。通常，使用虚拟方法。

覆盖与隐藏

如果方法声明指定的方法标识符和参数签名与继承方法相同，但不包括重写，则新声明仅隐藏继承的方法标识符和参数签名，而不重写它。这两种方法都存在于后代类中，其中方法名称是静态绑定的。例如：

```
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 指令

reintroducing 指令禁止显示有关隐藏以前声明的虚拟方法的编译器警告。例如：

```
procedure DoSomething; reintroduce; // The ancestor class also
                                    // has a DoSomething method
```

如果要使用新方法隐藏继承的虚拟方法，请使用重新引入。

抽象方法

抽象方法是虚拟方法或动态方法，在声明它的类中没有实现。它的实现被推迟到后代类。抽象方法必须在虚拟或动态之后使用指令 abstract 声明。例如：

```
procedure DoSomething; virtual; abstract;
```

只能在已重写抽象方法的类或类的实例中调用抽象方法。

类方法

大多数方法称为实例方法，因为它们对对象的单个实例进行操作。类方法是对类而不是对象进行操作的方法（构造函数除外）。类方法有两种类型：普通类方法和类静态方法。

普通类方法

方法的定义必须以保留字 class 开头。例如：

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    ...
  end;
```

类方法的定义声明也必须以 class 开头。例如：

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
    ...
end;
```

在类方法的定义声明中，标识符 Self 表示调用该方法的类（可以是定义该方法的类的后代）。如果在类 C 中调用该方法，则 Self 属于 C 的类型类。因此，不能使用 Self 访问实例字段、实例属性和普通（对象）方法。可以使用 Self 调用构造函数和其他类方法，或者访问类属性和类字段。

可以通过类引用或对象引用调用类方法。当通过对象引用调用它时，对象的类将成为 Self 的值。

类静态方法

与类方法一样，可以在没有对象引用的情况下访问类静态方法。与普通的类方法不同，类静态方法根本没有 Self 参数。他们也无法访问任何实例成员。（他们仍然可以访问类字段、类属性和类方法。此外，与类方法不同，类静态方法不能声明为虚拟方法。

通过将单词 static 附加到其声明中，方法被定为类静态，例如：

```
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;
```

与类方法一样，可以通过类类型调用类静态方法（例如，没有对象引用），例如：

```
TMyClass.X := 17;
TMyClass.StatProc('Hello');
```

方法重载

可以使用重载指令重新声明方法。在这种情况下，如果重新声明的方法具有与其祖先不同的参数签名，则会重载继承的方法而不隐藏它。在后代类中调用该方法将激活与调用中的参数匹配的任何实现。

如果重载虚拟方法，请在后代类中重新声明它时使用 reintroducing 指令。例如：

```
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
```

在一个类中，不能发布多个同名的重载方法。维护运行时类型信息需要为每个已发布成员指定唯一的名称：

```
type
    TSomeClass = class
      published
        function Func(P: Integer): Integer;
        function Func(P: Boolean): Integer;    // error
          ...
```

不能重载用作属性读取或写入说明符的方法。

重载方法的实现必须重复类声明中的参数列表。有关重载的详细信息，请参阅过程和函数中的《重载过程和函数 （Delphi）》。

构造函数

构造函数是创建和初始化实例对象的特殊方法。构造函数的声明看起来像过程声明，但它以构造函数一词开头。例子：

```
constructor Create;
constructor Create(AOwner: TComponent);
```

构造函数必须使用默认寄存器调用约定。尽管声明不指定返回值，但构造函数返回对它创建或调用的对象的引用。

一个类可以有多个构造函数，但大多数只有一个构造函数。通常将构造函数称为 Create。

若要创建对象，请对类类型调用构造函数方法。例如：

```
MyObject := TMyClass.Create;
```

这将为新对象分配存储，将所有序号字段的值设置为零，为所有指针和类类型字段分配 nil，并使所有字符串字段为空。接下来执行构造函数实现中指定的其他操作;通常，对象基于作为参数传递给构造函数的值进行初始化。最后，构造函数返回对新分配和初始化的对象的引用。返回值的类型与构造函数调用中指定的类类型相同。

如果在执行对类引用调用的构造函数期间引发异常，则会自动调用 Destroy 析构函数来销毁未完成的对象。

当使用对象引用（而不是类引用）调用构造函数时，它不会创建对象。相反，构造函数对指定的对象进行操作，仅执行构造函数实现中的语句，然后返回对该对象的引用。构造函数通常与继承的保留字一起在对象引用上调用，以执行继承的构造函数。

下面是类类型及其构造函数的示例：

```
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;
```

构造函数的第一个操作通常是调用继承的构造函数来初始化对象的继承字段。然后，构造函数初始化后代类中引入的字段。由于构造函数始终清除它为新对象分配的存储空间，因此所

有字段都以零（序号类型）、nil（指针和类类型）、空（字符串类型）或未分配（变体）的值开头。因此，除了非零或非空值外，无需初始化构造函数实现中的字段。

通过类类型标识符调用时，声明为 virtual 的构造函数等效于静态构造函数。但是，当与类引用类型结合使用时，虚拟构造函数允许对象的多态构造，即构造其类型在编译时未知的对象。《（请参阅类引用。）》

析构函数

析构函数是一种特殊方法，用于销毁调用它的对象并解除分配其内存。析构函数的声明看起来像过程声明，但它以析构函数一词开头。例：

```
destructor SpecialDestructor(SaveData: Boolean);
destructor Destroy; override;
```

Win32 上的析构函数必须使用默认寄存器调用约定。尽管一个类可以有多个析构函数，但建议每个类重写继承的 Destroy 方法，并且不声明其他析构函数。

若要调用析构函数，必须引用实例对象。例如：

```
MyObject.Destroy;
```

调用析构函数时，首先执行析构函数实现中指定的操作。通常，这些包括销毁任何嵌入对象和释放对象分配的资源。然后，释放为对象分配的存储。

下面是析构函数实现的示例：

```
destructor TShape.Destroy;
begin
    FBrush.Free;
    FPen.Free;
    inherited Destroy;
end;
```

析构函数实现中的最后一个操作通常是调用继承的析构函数来销毁对象的继承字段。

如果在创建对象期间引发异常，则会自动调用 Destroy 来释放未完成的对象。这意味着销毁必须准备好处置部分构造的对象。由于构造函数在执行其他操作之前将新对象的字段设置为零值或空值，因此部分构造对象中的类类型和指针类型字段始终为 nil。因此，析构函数应在对类类型或指针类型字段进行操作之前检查 nil 值。调用 Free 方法（在 TObject 中定义）而不是 Destroy 提供了一种在销毁对象之前检查 nil 值的便捷方法。

类构造函数

类构造函数是开发人员无法访问的特殊类方法。编译器会自动将对类构造函数的调用插入到定义类的单元的初始化部分。通常，类构造函数用于初始化类的静态字段或执行某种类型的初始化，这是类或任何类实例正常运行所必需的。尽管通过将类初始化代码放入初始化部分可

以获得相同的结果，类构造函数的好处是帮助编译器决定哪些类应包含在最终二进制文件中，哪些应从中删除。

下一个示例显示了初始化类字段的常用方法：

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

implementation

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

end.
```

此方法有一个很大的缺点：即使应用程序可以包含声明 TBox 的单元，它也可能永远不会实际使用 TBox 类。在当前示例中，TBox 类包含在生成的二进制文件中，因为它在初始化部分中被引用。若要缓解此问题，请考虑使用类构造函数：

```
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.
```

在这种情况下，编译器会检查 TBox 是否实际在应用程序中的任何位置使用，如果使用，则会自动将对类构造函数的调用添加到单元的初始化部分。

注意：尽管编译器负责对类的初始化进行排序，但在某些复杂情况下，排序可能会变得随机。当一个类的类构造函数依赖于另一个类的状态时，就会发生这种情况，而另一个类又依赖

于第一个类。

注意：泛型类或记录的类构造函数可以执行多次。在这种情况下，类构造函数的确切执行次数取决于泛型类型的专用版本数。例如，专用 TList 类的类构造函数<String>可以在同一应用程序中执行多次。

类析构函数

类析构函数与类构造函数相反，因为它们执行类的终结。类析构函数具有与类构造函数相同的优点，但最终目的除外。

下面的示例基于类构造函数中显示的示例构建，并介绍了终结例程：

```
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.
```

注意：泛型类或记录的类析构函数可能会执行多次。在这种情况下，类析构函数的确切执行次数取决于泛型类型的专用版本数。例如，专用 TList 类的类析构函数<String>可以在同一应用程序中执行多次。

消息方法

消息方法实现对动态调度消息的响应。消息方法语法在所有平台上都受支持。VCL 使用消息方法来响应 Windows 消息。

消息方法是通过在方法声明中包含 message 指令，后跟一个从 1 到 49151 的整数常量（指定消息 ID）来创建消息方法。对于 VCL 控件中的消息方法，整数常量可以是消息单元中定义的 Win32 消息 ID 之一，以及相应的记录类型。消息方法必须是采用单个 var 参数的过程。

例如：

```
type
    TTextBox = class(TCustomControl)
      private
        procedure WMChar(var Message: TWMChar); message WM_CHAR;
        ...
    end;
```

消息方法不必包含 override 指令即可重写继承的消息方法。实际上，它不必指定与其重写的方法相同的方法名称或参数类型。消息 ID 单独确定该方法响应哪条消息以及它是否是重写。

实现消息方法

消息方法的实现可以调用继承的消息方法，如以下示例所示：

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
    if Message.CharCode = Ord(#13) then
        ProcessEnter
    else
        inherited;
end;
```

继承的语句在类层次结构中向后搜索，并调用与当前方法具有相同 ID 的第一个消息方法，并自动将消息记录传递给它。如果没有祖先类为给定 ID 实现消息方法，则继承的将调用最初在 TObject 中定义的 DefaultHandler 方法。

在 TObject 中实现 DefaultHandler 只是返回而不执行任何操作。通过重写 DefaultHandler，类可以实现自己的默认消息处理。在 Win32 上，控件的 DefaultHandler 方法调用 Win32 API DefWindowProc。

消息分发

消息处理程序很少直接调用。相反，消息使用从 TObject 继承的调度方法调度到对象：

```
procedure Dispatch(var Message);
```

传递给调度的消息参数必须是记录，其第一个条目是包含消息 ID 的 Word 类型的字段。

Dispatch 在类层次结构中向后搜索（从调用它的对象的类开始），并为传递给它的 ID 调用第一个消息方法。如果未找到给定 ID 的消息方法，则调度将调用 DefaultHandler。

属性

关于属性

属性（如字段）定义对象的属性。但是，虽然字段只是一个存储位置，其内容可以检查和更改，但属性将特定操作与读取或修改其数据相关联。属性提供对对象属性的访问的控制，并允许计算属性。

属性的声明指定名称和类型，并至少包含一个访问说明符。属性声明的语法为：

property propertyName[indexes]: type index integerConstant specifiers;

其中：

- ✓ propertyName 是任何有效的标识符。
- ✓ [indexes] 是可选的，是用分号分隔的参数声明序列。每个参数声明都有 identifier1， ...，identifiern： 类型的形式。有关详细信息，请参阅下面的数组属性。
- ✓ type 必须是预定义或以前声明的类型标识符。也就是说，属性声明如属性 Num： 0..9 ... 无效。
- ✓ index integerConstant 子句是可选的。有关详细信息，请参阅下面的索引说明符。
- ✓ specifiers 是 read, write, stored, default (or nodefault), and implements 的说明符序列。每个属性声明必须至少有一个 read 或 write 说明符。

属性由其访问说明符定义。与字段不同，属性不能作为 var 参数传递，也不能将 @ 运算符应用于属性。原因是属性不一定存在于内存中。例如，它可以有一个从数据库中检索值或生成随机值的读取方法。

属性访问

每个属性都有一个读取说明符和/或写入说明符。这些称为访问说明符，它们的形式为：

read fieldOrMethod
write fieldOrMethod

其中 fieldOrMethod 是在与属性相同的类中声明的字段或方法的名称，或者在祖先类中声明。

- ✓ 如果在同一类中声明 fieldOrMethod，则必须在属性声明之前发生。如果在祖先类中声明，则必须从后代中可见;也就是说，它不能是在不同单元中声明的祖先类的私有字段或方法。

- ✓ 如果 fieldOrMethod 是一个字段，则它必须与属性的类型相同。
- ✓ 如果 fieldOrMethod 是一个方法，则它不能是动态的，如果是虚拟的，则不能重载。此外，已发布属性的访问方法必须使用默认寄存器调用约定。
- ✓ 在读取说明符中，如果 fieldOrMethod 是一个方法，则它必须是结果类型与属性类型相同的无参数函数。（索引属性或数组属性的访问方法除外。）
- ✓ 在写入说明符中，如果 fieldOrMethod 是一个方法，则它必须是采用与属性类型相同的单个值或 const 参数的过程（如果是数组属性或索引属性，则采用更多值或常量参数）。

例如，给定声明：

```
property Color: TColor read GetColor write SetColor;
```

GetColor 方法必须声明为：

```
function GetColor: TColor;
```

并且 SetColor 方法必须声明为以下方法之一：

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

（当然，SetColor 参数的名称不一定是 Value。

在表达式中引用属性时，将使用读取说明符中列出的字段或方法读取其值。在赋值语句中引用属性时，将使用写入说明符中列出的字段或方法写入其值。

下面的示例声明了一个名为 TCompass 的类，该类具有一个名为 Heading 的已发布属性。标题的值通过 FHeading 字段读取，并通过 SetHeader 过程写入：

```
type
    THeading = 0..359;
    TCompass = class(TControl)
      private
        FHeading: THeading;
        procedure SetHeading(Value: THeading);
      published
        property Heading: THeading read FHeading write SetHeading;
        ...
     end;
```

鉴于此声明，声明：

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

对应：

```
if Compass.FHeading = 180 then GoingSouth;
 Compass.SetHeading(135);
```

在 TCompass 类中，没有任何操作与读取标题属性相关联;读取操作包括检索存储在 FHeading 字段中的值。另一方面，将值分配给 Heading 属性将转换为对 SetHeading 方法的调用，该方法大概将新值存储在 FHeading 字段中以及执行其他操作。例如，SetHeader 可以像这样实现：

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint;       // update user interface to reflect new value
  end;
end;
```

声明仅包含读取说明符的属性是只读属性，声明仅包含写入说明符的属性是只写属性。将值分配给只读属性或在表达式中使用只写属性是错误的。

数组属性

数组属性是索引属性。它们可以表示列表中的项、控件的子控件和位图的像素等内容。
数组属性的声明包括一个参数列表，该列表指定索引的名称和类型。例如：

```
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;
```

索引参数列表的格式与过程或函数的参数列表的格式相同，只是参数声明括在括号而不是括号中。与只能使用序号类型索引的数组不同，数组属性允许任何类型的索引。
对于数组属性，访问说明符必须列出方法而不是字段。读取说明符中的方法必须是一个函数，该函数以相同的顺序获取属性的索引参数列表中列出的参数的数量和类型，并且其结果类型与属性的类型相同。写入说明符中的方法必须是一个过程，该过程采用属性的索引参数列表中列出的参数的数量和类型，顺序相同，再加上与属性类型相同的附加值或 const 参数。
例如，上述数组属性的访问方法可能声明为：

```
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);
```

通过索引属性标识符来访问数组属性。例如，语句：

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\BIN';
```

对应：

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\BIN');
```

数组属性的定义可以后跟默认指令，在这种情况下，数组属性将成为类的默认属性。例如：

```
type
    TStringArray = class
     public
        property Strings[Index: Integer]: string ...; default;
            ...
     end;
```

如果类具有默认属性，则可以使用缩写 object[index] 访问该属性，该属性等效于object.property[index]。例如，给定上面的声明，StringArray.Strings[7] 可以缩写为 StringArray[7]。一个类只能有一个具有给定签名的默认属性（数组参数列表），但可以重载默认属性。更改或隐藏后代类中的默认属性可能会导致意外行为，因为编译器始终静态绑定到属性。

索引说明符

索引说明符允许多个属性共享相同的访问方法，同时表示不同的值。索引说明符由指令索引后跟一个介于 -2147483647 和 2147483647 之间的整数常量组成。如果属性具有索引说明符，则其读取和写入说明符必须列出方法而不是字段。例如：

```
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;
```

具有索引说明符的属性的访问方法必须采用 Integer 类型的额外值参数。对于读取函数，它必须是最后一个参数;对于写入过程，它必须是倒数第二个参数（在指定属性值的参数之前）。当程序访问该属性时，该属性的整数常量将自动传递给访问方法。

鉴于上面的声明，如果矩形的类型是 TRectangle，则：

```
Rectangle.Right := Rectangle.Left + 100;
```

对应：

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

存储说明符

可选的 stored, default, and nodefault 指令称为存储说明符。它们对程序行为没有影响，但控制是否将已发布属性的值保存在窗体文件中。

存储的指令后必须跟 True、False、布尔字段的名称或返回布尔值的无参数方法的名称。例如：

```
property Name: TComponentName read FName write SetName stored False;
```

如果属性没有存储指令，则将其视为指定了存储的 True。

默认指令后必须跟一个与属性类型相同的常量。例如：

```
property Tag: Longint read FTag write FTag default 0;
```

若要覆盖继承的默认值而不指定新值，请使用 nodefault 指令。default 和 nodefault 指令仅支持序号类型和集合类型，前提是集合基类型的上限和下限的序号值介于 0 和 31 之间;如果声明此类属性时没有默认值或无默认值，则将其视为指定了 nodefault。对于实数、指针和字符串，隐式默认值分别为 0、nil 和 ''（空字符串）。

注： 不能将序号值 -2147483648 用作默认值。此值在内部用于表示 nodefault。

保存组件的状态时，将检查组件已发布属性的存储说明符。如果属性的当前值与其默认值不同（或者没有默认值），并且存储的说明符为 True，则保存属性的值。否则，不会保存属性的值。

注： 属性值不会自动初始化为默认值。也就是说，默认指令仅控制何时将属性值保存到窗体文件中，而不控制新创建的实例上属性的初始值。

数组属性不支持存储说明符。在数组属性声明中使用默认指令时具有不同的含义。请参阅上面的数组属性。

属性覆盖和重新声明

未指定类型的属性声明称为属性重写。属性替代允许您更改属性的继承可见性或说明符。最简单的重写仅包含保留字属性，后跟继承的属性标识符;此表单用于更改属性的可见性。例如，如果祖先类将属性声明为受保护，则派生类可以在该类的公共或已发布部分中重新声明该属性。属性覆盖可以包括读取、写入、存储、默认和无默认指令;任何此类指令都将覆盖相应的继承指令。重写可以替换继承的访问说明符、添加缺少的说明符或增加属性的可见性，但不能删除访问说明符或降低属性的可见性。重写可以包含 implements 指令，该指令添加到已实现接口列表中，而无需删除继承的接口。

以下声明说明了属性重写的使用：

```
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;
```

Size 的重写添加了一个写入说明符以允许修改属性。“文本”和“颜色”的替代会将属性的可见性从“受保护”更改为“已发布”。Color 的属性重写还指定如果属性的值不是 clBlue，则应提交该属性。

重新声明包含类型标识符的属性将隐藏继承的属性，而不是重写它。这意味着将创建一个与继承属性同名的新属性。指定类型的任何属性声明都必须是完整的声明，因此必须至少包含

一个访问说明符。

无论属性在派生类中是隐藏还是重写，属性查找始终是静态的。也就是说，用于标识对象的变量的声明（编译时）类型决定了其属性标识符的解释。因此，在执行以下代码后，读取或赋值给 MyObject.Value 会调用 Method1 或 Method2，即使 MyObject 包含 TDescendant 的实例也是如此。但是，您可以将 MyObject 强制转换为 TDescendant 以访问后代类的属性及其访问说明符：

```
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;
```

类属性

可以在没有对象引用的情况下访问类属性。类属性访问器本身必须声明为类静态方法或类字段。类属性是使用类属性关键字声明的。类属性不能发布，也不能具有存储或默认值定义。

通过使用类 var 块声明，可以在类声明中引入类静态字段块。在类 var 之后声明的所有字段都具有静态存储属性。类 var 块由以下项终止：

1. 另一个类变量声明
2. 过程或函数（即方法）声明（包括类过程和类函数）
3. 属性声明（包括类属性）
4. 构造函数或析构函数声明
5. 可见性范围说明符（公共、专用、受保护、已发布、严格私有和严格保护）

例如：

```
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;
```

您可以使用代码访问上述类属性：

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

事件

关于事件

事件将系统中的实例与响应该事件的代码链接在一起。该事件触发了称为事件处理程序的过程的执行。事件处理程序执行响应事件所需的任务。事件允许在设计时或运行时自定义组件的行为。若要更改组件的行为，请将事件处理程序替换为具有所需行为的自定义事件处理程序。

事件属性和事件处理程序

用 Delphi 编写的组件使用属性来指示事件发生时将执行的事件处理程序。按照约定，事件属性的名称以“On”开头，并且该属性是使用字段而不是读/写方法实现的。属性存储的值是一个方法指针，指向事件处理程序过程。

在下面的示例中，类包括一个 TPingEvent 类型的 OnPing 事件。FOnPing 字段用于存储事件处理程序。此示例中的事件处理程序 TListener.Ping 打印“TListener 已被 ping！”。

```
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.
```

触发多个事件处理程序

在 Delphi 中，只能为事件分配一个事件处理程序。如果必须执行多个事件处理程序以响应事件，则分配给该事件的事件处理程序必须调用任何其他事件处理程序。在下面的代码中，

一个名为 TListenerSubclass 的 TListener 子类有自己的名为 Ping2 的事件处理程序。在此示例中，Ping2 事件处理程序必须显式调用 TListener.Ping 事件处理程序，以便触发它以响应 OnPing 事件：

```
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.
```

类引用

有时，操作是对类本身执行的，而不是对类（即对象）的实例执行的。例如，当您使用类引用调用构造函数方法时，就会发生这种情况。您始终可以使用其名称引用特定类，但有时需要声明将类作为值的变量或参数，在这些情况下，您需要类引用类型。

类引用类型

类引用类型（有时称为元类）由以下形式的构造表示：

```
class of type
```

其中 type 是任何类类型。标识符类型本身表示其类型为类型类的值。如果 type1 是 type2 的祖先，则 type2 的类与 type1 的类赋值兼容。因此：

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

```
var AnyObj: TClass;
```

声明一个名为 AnyObj 的变量，该变量可以保存对任何类的引用。 （类引用类型的定义不能直接出现在变量声明或参数列表中。可以将值 nil 赋给任何类引用类型的变量。

若要了解如何使用类引用类型，请查看 System.Classes.TCollection 的构造函数声明（在 System.Classes 单元中）：

```
type TCollectionItemClass = class of TCollectionItem;
        ...
TCollection = class(TPersistent)
        ...
    constructor Create(ItemClass: TCollectionItemClass);
```

此声明指出，若要创建 TCollection 实例对象，必须将源自 TCollectionItem 的类的名称传递给构造函数。

如果要在编译时实际类型未知的类或对象上调用类方法或虚拟构造函数，类引用类型非常有用。

构造函数和类引用

可以使用类引用类型的变量调用构造函数。这允许构造在编译时类型未知的对象。例如：

```
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;
```

CreateControl 函数需要一个类引用参数来告诉它要创建的控件类型。它使用此参数调用类的构造函数。由于类类型标识符表示类引用值，因此对 CreateControl 的调用可以指定要为其创建实例的类的标识符。例如：

```
CreateControl（TEdit， 'Edit1'， 10， 10， 100， 20）;
```

使用类引用调用的构造函数通常是虚拟的。调用激活的构造函数实现取决于类引用的运行时类型。

类运算符

类方法对类引用进行操作。每个类都从 TObject 继承了两个类方法，称为 ClassType 和 ClassParent。这些方法分别返回对对象的类和对象的直接祖先类的引用。这两种方法都返回一个 TClass 类型的值（其中 TClass = TObject 类），可以将其转换为更具体的类型。每个类还继承一个名为 InheritsFrom 的方法，该方法测试调用它的对象是否来自指定的类。这些方法由 is 和 as 运算符使用，很少需要直接调用它们。

is 运算符

执行动态类型检查的 is 运算符用于验证对象的实际运行时类。表达式：

object is class

如果对象是由类或其后代之一表示的类的实例，则返回 True，否则返回 False。（如果对象为 nil，则结果为 False。如果声明的对象类型与类无关（即，如果类型不同且一个不是另一个的祖先），则会产生编译错误。例如：

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

此语句将 ActiveControl 变量强制转换为 TEdit 类型。首先，它验证 ActiveControl 引用的对象是 TEdit 的实例还是其后代之一。

as 运算符

as 运算符执行已检查的类型转换。表达式

object as class

返回对与 Object 相同的对象的引用，但具有类给出的类型。在运行时，对象必须是类或其后代之一表示的类的实例，或者为 nil;否则将引发异常。如果声明的对象类型与类无关（即，如果类型不同且一个不是另一个的祖先），则会产生编译错误。例如：

```
with Sender as TButton do
  begin
    Caption := '&Ok';
    OnClick := OkClick;
  end;
```

运算符优先级规则通常要求将类型转换括在括号中。例如：

```
(Sender as TButton).Caption := '&Ok';
```

异常

关于异常

当错误或其他事件中断程序的正常执行时，将引发异常。异常将控制权转移到异常处理程序，该处理程序允许您将正常程序逻辑与错误处理分开。由于异常是对象，因此可以使用继承将它们分组到层次结构中，并且可以在不影响现有代码的情况下引入新的异常。异常可以携带信息（如错误消息），从引发它的位置到处理它的点。

当应用程序使用 SysUtils 单元时，大多数运行时错误会自动转换为异常。可以捕获和处理许多本来会终止应用程序的错误，例如内存不足、被零除和一般保护错误。

何时使用异常

异常提供了一种优雅的方式来捕获运行时错误，而无需停止程序，也没有尴尬的条件语句。异常处理语义强加的要求会降低代码/数据大小和运行时性能。虽然几乎可以出于任何原因引发异常，并通过将其包装在 try...except 或 try...finally 中来保护几乎任何代码块，在实践中，这些工具最好保留给特殊情况。

异常处理适用于发生几率低或难以评估，但其后果可能是灾难性的错误（例如应用程序崩溃）;对于复杂或难以在 IF 中测试的错误条件...然后语句;以及当您需要响应操作系统或源代码不受控制的例程引发的异常时。异常通常用于硬件、内存、I/O 和操作系统错误。

条件语句通常是测试错误的最佳方法。例如，假设您要在尝试打开文件之前确保该文件存在。你可以这样做：

```
try
    AssignFile(F, FileName);
    Reset(F);        // raises an EInOutError exception if file is not found
except
    on Exception do ...
end;
```

但是，您也可以通过使用以下命令来避免异常处理的开销：

```
if FileExists(FileName) then       // returns False if file is not found; raises no exception

begin
    AssignFile(F, FileName);
    Reset(F);
```

```
end;
```

断言提供了另一种在源代码中任何位置测试布尔条件的方法。当 Assert 语句失败时，程序要么因运行时错误而停止，要么（如果它使用 SysUtils 单元）引发 SysUtils.EAssertionFailed 异常。断言应仅用于测试您不希望发生的条件。

声明异常类型

异常类型的声明方式与其他类一样。实际上，可以使用任何类的实例作为异常，但建议从 SysUtils 中定义的 SysUtils.Exception 类派生异常。

您可以使用继承将异常分组到族中。例如，SysUtils 中的以下声明为数学错误定义了一系列异常类型：

```
type
    EMathError = class(Exception);
    EInvalidOp = class(EMathError);
    EZeroDivide = class(EMathError);
    EOverflow = class(EMathError);
    EUnderflow = class(EMathError);
```

给定这些声明，您可以定义单个 SysUtils.EMathError 异常处理程序，该处理程序还处理 SysUtils.EInvalidOp、SysUtils.EZeroDivide、SysUtils.Overflow 和 SysUtils.EUnderflow。

异常类有时定义传达有关错误的其他信息的字段、方法或属性。例如：

```
type EInOutError = class(Exception)
        ErrorCode: Integer;
    end;
```

引发和处理异常

若要引发异常对象，请将异常类的实例与 raise 语句一起使用。例如：

```
raise EMathError.Create;
```

一般来说，引发异常的形式是

```
raise object at address
```

其中对象和 at 地址都是可选的。指定地址时，它可以是计算结果为指针类型的任何表达式，但通常是指向过程或函数的指针。例如：

```
raise Exception.Create('Missing parameter') at @MyFunction;
```

使用此选项可从堆栈中比实际发生错误的点更早的点引发异常。

引发异常（即在 raise 语句中引用）时，它由特殊的异常处理逻辑控制。raise 语句永远不会以正常方式返回控制权。相反，它将控制权转移到可以处理给定类的异常的最内层异常处理程序。 （最内层的处理程序是其 try...except 最近进入但尚未退出的块。）

例如，下面的函数将字符串转换为整数，如果结果值超出指定范围，则会引发 SysUtils.ERangeError 异常。

```
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;
```

请注意在 raise 语句中调用的 CreateFmt 方法。SysUtils.Exception 及其后代具有特殊的构造函数，这些构造函数提供了创建异常消息和上下文 ID 的替代方法。

引发的异常在处理后会自动销毁。切勿尝试手动销毁引发的异常。

注意：在单元的初始化部分引发异常可能不会产生预期的结果。正常的异常支持来自 SysUtils 单元，必须先初始化该单元，然后才能获得此类支持。如果在初始化期间发生异常，则会完成所有已初始化的单元（包括 SysUtils），并重新引发异常。然后捕获并处理异常，通常是通过中断程序。同样，如果在引发异常时 SysUtils 已经完成，则在单元的最终确定部分中引发异常可能不会产生预期的结果。

try...except 语句

异常在 try...except 语句中处理。例如：

```
try
    X := Y/Z;
    except
      on EZeroDivide do HandleZeroDivide;
end;
```

此语句尝试将 Y 除以 Z，但如果引发 SysUtils.EZeroDivide 异常，则会调用名为 HandleZeroDivide 的例程。

try...except 语句的语法如下：

```
try statements except exceptionBlock end
```

其中 statements 是语句序列（由分号分隔），exceptionBlock 是以下其一：

- ✓ 另一个语句序列或
- ✓ 异常处理程序序列，可选后跟

else statements

异常处理程序的格式为：

on identifier: type do statement

其中 identifier：是可选的（如果包含，标识符可以是任何有效的标识符），type 是用于表示异常的类型，statement 是任何语句。

try...except 语句执行初始语句列表中的语句。如果未引发异常，则忽略异常块（exceptionBlock），并将控制权传递到程序的下一部分。

如果在执行初始语句列表期间引发异常，无论是通过语句列表中的 raise 语句还是由从语句列表中调用的过程或函数引发，则会尝试“处理”异常：

- ✓ 如果异常块中的任何处理程序与异常匹配，则控制权将传递给第一个此类处理程序。异常处理程序“匹配”异常，以防处理程序中的类型是异常的类或该类的祖先。
- ✓ 如果未找到此类处理程序，则控制权将传递给 else 子句中的语句（如果有）。
- ✓ 如果异常块只是没有任何异常处理程序的语句序列，则控制权将传递给列表中的第一个语句。

如果上述条件均不满足，则在最近输入的下一个 try...except 块中继续搜索。除了尚未退出的语句。如果在那里找不到合适的处理程序、else 子句或语句列表，则搜索将传播到下一个最近输入的 try...except 语句，等等。如果最外面的 try...except 语句到达，但仍未处理异常，否则程序将终止。

处理异常时，堆栈将追溯到包含 try...except 发生处理的语句，并且控制权将转移到执行的异常处理程序、else 子句或语句列表。此过程将丢弃在进入 try 后发生的所有过程和函数调用。处理异常的 except 语句。然后，通过调用其 Destroy 析构函数自动销毁异常对象，并将控制权传递给 try...except 后续声明。（如果对 Exit, Break, or Continue 标准过程的调用导致控件离开异常处理程序，则仍会自动销毁异常对象。

在下面的示例中，第一个异常处理程序处理除以零的异常，第二个处理溢出异常，最后一个处理所有其他数学异常。SysUtils.EMathError 在异常块中排在最后，因为它是其他两个异常类的祖先;如果它首先出现，则永远不会调用其他两个处理程序：

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

异常处理程序可以在异常类的名称之前指定标识符。这将声明 on...do，以在执行后续语

句期间表示异常对象。标识符的范围仅限于该语句。例如：

```
try
  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

如果异常块指定了 else 子句，则 else 子句将处理块的异常处理程序未处理的任何异常。例如：

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

在这里，else 子句处理任何不是 SysUtils.EMathError 的异常。
不包含异常处理程序，而仅包含语句列表的异常块处理所有异常。例如：

```
try
   ...
except
    HandleException;
end;
```

在这里，HandleException 例程处理由于在 try 和 exclude 之间执行语句而发生的任何异常。

重新引发异常

当保留字 raise 在异常块中时：

- ✓ 使用普通引发重新引发当前异常对象。这允许异常处理程序以有限的方式响应错误，然后重新引发异常。当过程或函数在发生异常后必须清理但无法完全处理异常时，重新引发很有用。例如：

```
try
  raise Exception.Create('Error Message');
```

```
except
  on E: Exception do
  begin
    E.Message := E.Message +sLineBreak+ 'FATAL';
    raise;
  end;
end;
```

✓ 另一方面，不要使用 raise E 重新引发当前异常对象，只需使用 raise 代替。否则可能会发生访问冲突。这是不正确的：

```
try
  raise Exception.Create('Error Message');
except
  on E: Exception do
  begin
    E.Message := E.Message +sLineBreak+ 'FATAL';
    raise E;
  end;
end;
```

例如，GetFileList 函数分配一个 TStringList 对象，并用与指定搜索路径匹配的文件名填充该对象：

```
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 创建一个 TStringList 对象，然后使用 FindFirst 和 FindNext 函数（在 SysUtils

中定义）对其进行初始化。如果初始化失败（例如，因为搜索路径无效，或者因为没有足够的内存来填充字符串列表），GetFileList 需要释放新的字符串列表，因为调用方还不知道它的存在。因此，字符串列表的初始化是在 try...except 声明。如果发生异常，语句的异常块将释放字符串列表，然后重新引发异常。

嵌套异常

在异常处理程序中执行的代码本身可以引发和处理异常。只要这些异常也在异常处理程序中处理，它们就不会影响原始异常。但是，一旦异常处理程序中引发的异常传播到该处理程序之外，原始异常就会丢失。下面的 Tan 函数对此进行了说明：

```
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;
```

如果在执行 Tan 期间发生 SysUtils.EMathError 异常，则异常处理程序会引发 ETrigError。由于 Tan 没有为 ETrigError 提供处理程序，因此异常会传播到原始异常处理程序之外，从而导致 SysUtils.EMathError 异常被销毁。对于调用方来说，Tan 函数似乎引发了 ETrigError 异常。

try...finally 语句

有时，您希望确保操作的特定部分已完成，无论操作是否因异常而中断。例如，当例程获得对资源的控制权时，释放资源通常很重要，而不管该例程是否正常终止。在这些情况下，您可以使用 try...finally 声明。

下面的示例演示打开和处理文件的代码如何确保文件最终关闭，即使在执行过程中发生错误也是如此：

```
Reset(F);
try
    ... // process file F
finally
    CloseFile(F);
end;
```

try...finally 语句的语法如下：

try statementList1 finally statementList2 end

其中每个语句列表是由分号分隔的语句序列。Try...finally 语句执行 statementList1（try 子句）中的语句。如果 statementList1 完成而不引发异常，则执行 statementList2 （最后子句）。如果在执行 statementList1 期间引发异常，则控制权将转移到 statementList2；一旦 statementList2 完成执行，就会重新引发异常。如果对 Exit, Break, or Continue 过程的调用导致控件离开 statementList1 ，则会自动执行 statementList2 。因此，无论 try 子句如何终止，都始终执行 finally 子句。

如果引发了异常但未在 finally 子句中处理，则该异常将从 try...finally 语句中传播出去，并且 try 子句中已引发的任何异常都将丢失。因此，finally 子句应处理所有本地引发的例外，以免干扰其他例外的传播。

标准异常类和例程

SysUtils 和 System 单元声明了几个用于处理异常的标准例程，包括 ExceptObject、ExceptAddr 和 ShowException。SysUtils、System 和其他单元还包括数十个异常类，所有这些类（除了 OutlineError）都源自 SysUtils.Exception。

类具有名为 Message 和 HelpContext 的属性，可用于传递上下文相关联机文档的错误说明和上下文 ID。它还定义了各种构造函数方法，这些方法允许您以不同的方式指定说明和上下文 ID。

类型和记录助手类

关于类和记录帮助程序

类或记录帮助程序是一种类型，当与另一个类或记录关联时，引入可在关联类型（或其后代）的上下文中使用的其他方法名称和属性。帮助程序是一种在不使用继承的情况下扩展类的方法，这对于根本不允许继承的记录也很有用。帮助程序只是为编译器在解析标识符时引入更广泛的范围。声明类或记录帮助程序时，需要声明帮助程序名称以及要使用帮助程序扩展的类型的名称。您可以在任何可以合法使用扩展类或记录的地方使用帮助程序。然后，编译器的解析范围将成为原始类型以及帮助程序。

类和记录帮助程序提供了一种扩展类型的方法，但不应将它们视为开发新代码时使用的设计工具。对于新代码，应始终依赖普通的类继承和接口实现。

帮助程序语法

声明类帮助程序的语法为：

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

```
    memberList
  end;
```

ancestor list 是可选的。它只能为类帮助程序指定，它指示由当前声明扩展的现有类帮助程序。

帮助程序类型不能声明实例数据，但允许使用类字段。

可见性范围规则和成员列表语法与普通类和记录类型相同。

注意：类和记录帮助程序不支持运算符重载。

您可以定义多个帮助程序并将其与单个类型关联。但是，只有零个或一个帮助程序适用于源代码中的任何特定位置。将应用在最接近的作用域中定义的帮助程序。类或记录帮助程序作用域以正常的 Delphi 方式确定（例如，在单元的 use 子句中从右到左）。

使用助手

下面的代码演示了类帮助程序的声明（记录帮助程序的行为方式相同）：

```
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
```

提示： 请记住，调用类帮助程序函数 MyFunc，因为类帮助程序优先于实际类类型。类和记录帮助程序可用于扩展内部类型和枚举类型。例如：

```
type
    TMyEnum = (one, two, three);
    TMyEnumHelper = record helper for TMyEnum
      function GetNext: TMyEnum;
    end;
```

内嵌类型声明

类型声明可以嵌套在类声明中。嵌套类型通常在整个面向对象编程中使用。它们提供了一种将概念上相关的类型保持在一起并避免名称冲突的方法。用于声明嵌套类型的相同语法可以与 Win32 Delphi 编译器一起使用。

声明嵌套类型

嵌套类型声明遵循数据类型、变量和常量索引 （Delphi） 中定义的类型声明语法。

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

```
        type
            nestedTypeDeclaration

        memberList
    end;
```

嵌套类型声明由非标识符标记（例如过程、类、类型和所有可见性范围说明符）的第一次出现终止。

常规辅助功能规则适用于嵌套类型及其包含类型。嵌套类型可以访问其容器类的实例变量（字段、属性或方法），但它必须具有对象引用才能执行此操作。嵌套类型可以在没有对象引用的情况下访问类字段、类属性和类静态方法，但正常的 Delphi 可见性规则适用。

嵌套类型不会增加包含类的大小。创建包含类的实例不会同时创建嵌套类型的实例。嵌套类型仅通过其声明的上下文与其包含类相关联。

声明和访问嵌套类

下面的示例演示如何声明和访问嵌套类的字段和方法：

```
type
        TOuterClass = class
          strict private
              myField: Integer;

          public
              type
                  TInnerClass = class
                    public
                        myInnerField: Integer;
                        procedure innerProc;
                    end;

              procedure outerProc;
          end;
```

若要实现内部类的 innerProc 方法，必须使用外部类的名称限定其名称。例如：

```
procedure TOuterClass.TInnerClass.innerProc;
begin
    ...
end;
```

若要访问嵌套类型的成员，请使用与常规类成员访问一样的点表示法。例如：

```
var
    x: TOuterClass;
    y: TOuterClass.TInnerClass;

begin
    x := TOuterClass.Create;
    x.outerProc;
    ...
    y := TOuterClass.TInnerClass.Create;
    y.innerProc;
```

嵌套常量

可以在类类型中以与嵌套类型节相同的方式声明常量。常量节由与嵌套类型节相同的标记终止，具体而言，保留字或可见性说明符。不支持类型化常量，因此不能声明值类型的嵌套常量，例如 System.Currency 或 System.TDateTime。

嵌套常量可以是任何简单类型：序号、序号子范围、枚举、字符串和实数类型。

下面的代码演示嵌套常量的声明：

```
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.
```

操作符重载

关于运算符重载

Delphi 允许某些函数或“运算符”在记录声明中重载。运算符函数的名称映射到源代码中的符号表示形式。例如，Add 运算符映射到 + 符号。

编译器生成对相应重载的调用，将上下文（即返回类型和调用中使用的参数类型）与运算符函数的签名相匹配。

下表显示了可以重载的 Delphi 运算符：

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

除表中列出的运算符外，不得在记录上定义其他运算符。

重载运算符方法不能在源代码中按名称引用。要访问特定记录的特定运算符方法，请参阅：代码示例：OpOverloads_（Delphi）。对于以单词“operator”开头的方法列表中的语言记录，包含运算符标识符（例如：System.AnsiStringBase Methods）。您可以在自己的记录中实现上述任何运算符。

编译器将使用运算符作为记录，前提是：

对于二元运算符，输入参数之一必须是记录类型。

对于一元运算符，输入参数或返回值必须是记录类型。

对于使用相同符号的逻辑运算符和按位运算符，仅当操作数为布尔值时才使用逻辑运算符。由于此记录运算符的记录类型不是布尔值，因此仅当另一个操作数是布尔值时才使用逻辑运算符。

不对操作的分布或交换属性进行任何假设。对于二元运算符，第一个参数始终是左操作数，第二个参数始终是右操作数。在没有明确括号的情况下，假定关联性是从左到右的。

运算符方法的解析是通过操作中使用的类型的可访问运算符的联合完成的（请注意，这包括继承的运算符）。对于涉及两个不同类型 A 和 B 的操作，如果类型 A 具有隐式转换为 B，而 B 具有隐式转换为 A，则会发生歧义。隐式转换应仅在绝对必要时提供，并且应避免反身性。最好让类型 B 隐式地将自身转换为类型 A，而让类型 A 不知道类型 B（反之亦然）。

作为一般规则，运算符不应修改其操作数。相反，返回一个新值，该值是通过对参数执行操作来构造的。

重载运算符最常用于记录（即值类型）。
注： 记录帮助程序不支持运算符重载。

声明运算符重载

运算符重载在记录中声明，语法如下：

```
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;
```

重载运算符的实现还必须包括类运算符语法：

```
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;
```

以下是重载运算符的一些示例：

```
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;
```

7. 标准过程和输入输出

这些主题讨论文本和文件 I/O，并总结标准库例程。此处列出的许多过程和函数都是在 System 和 SysInit 单元中定义的，这些单元隐式用于每个应用程序。其他则内置于编译器中，但被视为在系统单元中。

一些标准例程采用诸如 SysUtils 之类的单元，必须在 use 子句中列出才能使它们在程序中可用。但是，不能在 use 子句中列出 System，也不应修改 System 单元或尝试显式重建它。

注意：对于新程序，您可能希望使用 System.Classes 和 System.SysUtils 单元中的文件管理类和函数。System.Classes.TStream 及其后代类目前建议用于 Delphi 中的常规文件处理（有关相关例程，请参阅 Streams、Reader 和 Writers）。对于文本文件处理，建议使用 TStreamReader 和 TStreamWriter，而不是调用 Write 和 Writeln。API 类别索引包含相关例程和类的列表。

注意：BlockRead 和 BlockWrite 具有非类型化的参数，这可能是内存损坏的根源。这两种方法都取决于记录大小的设置，该设置由先前的重置或重写调用隐式进行。使用 Streams 为程序员提供了更高级别的灵活性和功能。

文件输入输出

下表列出了输入和输出例程。

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.

文件变量是类型为 file 类型的任何变量。文件有三类：类型化、文本和非类型化。文件类型中给出了声明文件类型的语法。请注意，文件类型仅在 Win32 平台上可用。

在使用文件变量之前，必须通过调用 AssignFile 过程将其与外部文件相关联。外部文件通常是命名磁盘文件，但它也可以是设备，例如键盘或显示器。外部文件存储写入文件的信息或提供从文件中读取的信息。

建立与外部文件的关联后，必须打开文件变量以准备输入或输出。可以通过重置过程打开现有文件，也可以通过重写过程创建和打开新文件。使用“重置”打开的文本文件是只读的，使用“重写”和“追加”打开的文本文件是只写的。类型化文件和非类型化文件始终允许读取和写入，无论它们是使用“重置”还是“重写”打开的。

每个文件都是组件的线性序列，每个组件都具有文件的组件类型（或记录类型）。组件从零开始编号。

文件通常按顺序访问。也就是说，当使用标准过程读取或使用标准过程 Read 读取或使用标准过程 Write 写入组件时，当前文件位置将移动到下一个按数字顺序排列的文件组件。类型化文件和非类型化文件也可以通过标准过程 Seek 随机访问，该过程将当前文件位置移动到指定的组件。标准函数 FilePos 和 FileSize 可用于确定当前文件位置和当前文件大小。

当程序完成文件处理后，必须使用标准过程 CloseFile 关闭该文件。关闭文件后，将更新其关联的外部文件。然后，可以将文件变量与另一个外部文件相关联。

默认情况下，将自动检查对标准 I/O 过程和函数的所有调用是否存在错误，如果发生错误，则会引发异常（如果未启用异常处理，则终止程序）。可以使用 {$I+} 和 {$I-} 编译器指令打开和关闭此自动检查。当 I/O 检查关闭时，即，当过程或函数调用在 {$I-} 状态下编译时，I/O 错误不会导致引发异常;若要检查 I/O 操作的结果，必须改为调用标准函数 IOResult。

必须调用 IOResult 函数来清除错误，即使对该错误不感兴趣也是如此。如果未清除错误，并且 {$I-} 是当前状态，则下一个 I/O 函数调用将失败，并显示延迟的 IOResult 错误。

文本文件

本节总结了使用标准类型文本的文件变量的 I/O。

打开文本文件时，外部文件以特殊方式解释：它被视为表示格式化为行的字符序列，其中每行都由行尾标记（回车符，可能后跟换行符）终止。文本类型不同于 Char 的类型文件。

对于文本文件，有特殊形式的读取和写入，可用于读取和写入不属于 Char 类型的值。此类值会自动转换到其字符表示形式或从其字符表示形式转换。例如，Read（F， I），其中 I 是 Integer 类型变量，读取数字序列，将该序列解释为十进制整数，并将其存储在 I 中。

有两个标准文本文件变量：System.Input 和 System.Output 标准文件变量 System.Input 是与操作系统的标准输入（通常是键盘）关联的只读文件。标准文件变量 System.Output 是与操作系统的标准输出（通常是显示器）关联的只写文件。在应用程序开始执行之前，将自动打开 System.Input 和 System.Output，就像执行以下语句一样：

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

注：对于 Win32 应用程序，面向文本的 I/O 仅在控制台应用程序中可用，即使用在“项目选项”对话框的“链接”页上选中的“生成控制台应用程序”选项或使用 -cc 命令行编译器选项进行编译的应用程序。在 GUI（非控制台）应用程序中，任何使用 System.Input 或 System.Output 进行读取或写入的尝试都将产生 I/O 错误。

某些处理文本文件的标准 I/O 例程不需要将文件变量显式指定为参数。如果省略 file 参数，则默认采用 System.Input 或 System.Output，具体取决于过程或函数是面向输入还是面向输出。例如，Read（X） 对应于 Read（Input， X），Write（X） 对应于 Write（Output， X）。

如果在调用处理文本文件的输入或输出例程之一时指定了文件，则必须使用 AssignFile 将该文件与外部文件关联，并使用重置、重写或追加打开该文件。如果使用 Reset 打开的文件传递到面向输出的过程或函数，则会发生错误。如果使用“重写”或“追加”打开的文件传递到面向输入的过程或函数，也会发生错误。

无类型文件

非类型化文件是低级 I/O 通道，主要用于直接访问磁盘文件，而不考虑类型和结构。非类型化文件是用 word file 声明的，仅此而已。例如：

```
var DataFile: file;
```

对于非类型化文件，重置和重写过程允许使用额外的参数来指定数据传输中使用的记录大小。由于历史原因，默认记录大小为 128 字节。记录大小 1 是唯一正确反映任何文件确切大小的值。（当记录大小为 1 时，无法进行部分记录。

除“读取”和“写入”之外，还允许对非类型化文件使用所有类型化文件标准过程和函数。代替读取和写入，两个称为 BlockRead 和 BlockWrite 的过程用于高速数据传输。

文本文件设备驱动

您可以为程序定义自己的文本文件设备驱动程序。文本文件设备驱动程序是一组四个函数，完全实现了 Delphi 文件系统和某些设备之间的接口。

定义每个设备驱动程序的四个函数是“打开”、“传入”、“刷新”和“关闭”。每个函数的函数头为：

```
function DeviceFunc(var F: TTextRec): Integer;
```

其中 DeviceFunc 是函数的名称（即“打开”、“传入”、“刷新”或“关闭”）。设备接口函数的返回值将成为 IOResult 返回的值。如果返回值为零，则操作成功。

若要将设备接口函数与特定文件相关联，必须编写自定义的 Assign 过程。Assign 过程必须将四个设备接口函数的地址分配给文本文件变量中的四个函数指针。此外，它应该在模式字段中存储 fmClosed 魔术常量，将文本文件缓冲区的大小存储在 BufSize 中，在 BufPtr 中存储指向文本文件缓冲区的指针，并清除 Name 字符串。

例如，假设四个设备接口函数称为 DevOpen、DevInOut、DevFlush 和 DevClose，则 Assign 过程可能如下所示：

```
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;
```

“代码页”字段必须设置为“默认系统代码页”，RTL 才能使用设备接口函数。这些设备接口函数必须执行所需的任何特殊字符处理。

设备接口函数可以使用文件记录中的 UserData 字段来存储私人信息。产品文件系统在任何时候都不会修改此字段。

Open 函数

打开函数由重置、重写和追加标准过程调用 Open 函数，以打开与设备关联的文本文件。输入时，“模式”字段包含 fmInput、fmOutput 或 fmInOut，以指示是否从重置、重写或追加调用 Open 函数。

Open 函数根据 Mode 值准备文件以进行输入或输出。如果 Mode 指定了 fmInOut（指示从追加调用了 Open），则必须在 Open 返回之前将其更改为 fmOutput。

打开始终在任何其他设备接口函数之前调用。出于这个原因，AssignDev 只初始化 OpenFunc 字段，将其余向量的初始化留给 Open。然后，根据模式，Open 可以安装指向面向输入或输出的函数的指针。这样可以避免 InOut、Flush 函数和 CloseFile 过程确定当前模式。

InOut 函数

每当需要从设备输入或输出时，由 Read、Readln、Writeln、Eof、Eoln、SeekEof、SeekEoln 和 CloseFile 标准例程调用 InOut 函数。

当 Mode 为 fmInput 时，InOut 函数将最多 BufSize 字符读入 BufPtr^，并返回在 BufEnd 中读取的字符数。此外，它在 BufPos 中存储零。如果 InOut 函数在 BufEnd 中作为输入请求的结果返回零，则文件的 Eof 将变为 True。

当 Mode 为 fmOutput 时，InOut 函数从 BufPtr^ 写入 BufPos 字符，并在 BufPos 中返回零。

Flush 函数

刷新函数在每个读取、读取、写入和写入的末尾调用。它可以选择刷新文本文件缓冲区。

如果模式为 fmInput，则刷新函数可以在 BufPos 和 BufEnd 中存储零，以刷新缓冲区中剩余（未读）字符。此功能很少使用。

如果 Mode 为 fmOutput，则 Flush 函数可以像 InOut 函数一样写入缓冲区的内容，从而确保写入设备的文本立即显示在设备上。如果 Flush 不执行任何操作，则在缓冲区已满或文件关闭之前，文本不会显示在设备上。

Close 函数

关闭函数由 CloseFile 标准过程调用，用于关闭与设备关联的文本文件。（如果重置、重写和追加过程打开的文件已打开，则它们也会调用“关闭”。如果 Mode 为 fmOutput，则在调用 Close 之前，文件系统会调用 InOut 函数以确保所有字符都已写入设备。

处理 null 结尾的字符串

Delphi 语言的扩展语法允许将 Read、Readln、Str 和 Val 标准过程应用于从零开始的字符数组，并允许将 Write、Writeln、Val、AssignFile 和 Rename 标准过程应用于从零开始的字符数组和字符指针。

null 结尾的字符串函数

以下函数用于处理以 null 结尾的字符串。

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.

标准字符串处理函数具有启用多字节的对应项，这些对应函数还实现特定于区域设置的字符排序。多字节函数的名称以 Ansi- 开头。例如，StrPos 的多字节版本是 AnsiStrPos。多字节字符支持取决于操作系统，并基于当前区域设置。

宽字符字符串

系统单元提供三个函数：WideCharToString、WideCharLenToString 和 StringToWideChar，可用于将以 null 结尾的宽字符串转换为单字节或双字节长字符串。

赋值也将在字符串之间转换。例如，以下内容都有效：

```
MyAnsiString := MyWideString;
MyWideString := MyAnsiString;
```

其它标准过程

下表列出了系统库中常用的过程和函数。这不是标准例程的全部清单。

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.

8. 库和包

本节介绍如何在 Delphi 中创建静态和动态可加载的库。

注意：库在可以导出的内容方面比包受到的限制要大得多。库无法导出常量、类型和变量。也就是说，库中定义的类类型将不会在使用该库中的程序中看到。

若要导出简单过程和函数以外的项，建议使用包。仅当需要与其他编程的互操作性时，才应考虑库。

库和包

可动态加载库是 Windows 上的动态链接库 （DLL）、Mac 上的 DYLIB 或 Linux 上的共享对象 （SO）。它是可由应用程序和其他 DLL 或共享对象调用的例程的集合。与单元一样，可动态加载的库包含可共享的代码或资源。但是这种类型的库是一个单独编译的可执行文件，在运行时链接到使用它的程序。

Delphi 程序可以调用用其他语言编写的 DLL 和共享对象，用其他语言编写的应用程序可以调用用 Delphi 编写的 DLL 或共享对象。

调用动态可加载库

可以调用未链接到应用程序的操作系统例程。这些例程通常位于 DLL 或共享对象中。在 Windows 和 macOS 上，没有对导入例程的尝试进行编译时验证。这意味着编译程序时不需要存在该库。

在其他平台（如 Linux）上，要解析外部引用，您必须链接到共享对象。如果要避免验证，请使用 LoadLibrary 和 GetProcAddress，如动态加载部分中所述。

在调用 DLL 或共享对象中定义的例程之前，必须导入它们。这可以通过两种方式完成：通过声明外部过程或函数，或者通过直接调用操作系统。无论使用哪种方法，例程在运行时之前都不会链接到应用程序。

Delphi 不支持从 DLL 或共享对象导入变量。

静态加载

导入过程或函数的最简单方法是使用外部指令声明它。例如：

```
procedure DoSomething; external 'MYLIB.DLL';
```

如果在程序中包含此声明，则 MYLIB.DLL 将在程序启动时加载一次。在整个程序执行过程中，标识符 DoSomething 始终引用同一共享库中的同一入口点。

导入例程的声明可以直接放置在调用它们的程序或单元中。但是，为了简化维护，您可以将外部声明收集到单独的“导入单元”中，该单元还包含与库接口所需的任何常量和类型。使用导入单元的其他模块可以调用其中声明的任何例程。

延迟加载（仅限 Windows）

延迟指令可用于修饰外部例程，以延迟加载包含该例程的库。实际加载发生在首次调用例程时。以下示例演示了延迟指令的使用：

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

在上面的示例中，GetSomething 例程是从 somelibrary.dll 库中导入的。延迟指令确保 somelibrary.dll 不是静态链接到应用程序，而是动态链接。

延迟指令在运行应用程序的目标操作系统上不存在导入例程的情况下很有用。静态导入的例程要求操作系统在启动应用程序时查找并加载库。如果在加载的库中找不到例程，或者该库不存在，操作系统将停止应用程序的执行。使用延迟指令可以在运行时检查操作系统是否支持所需的 API;只有这样，您才能调用导入的例程。

延迟指令的另一个潜在用途与应用程序的内存占用有关：装饰不太可能使用的例程，因为延迟可能会减少应用程序的内存占用，因为库仅在需要时加载。滥用延迟可能会损害程序的速度性能（正如最终用户所感知的那样）。

注意：尝试调用无法解析的延迟例程会导致运行时错误（如果加载了 SysUtils 单元，则会出现异常）。

为了微调 Delphi 运行时库使用的延迟加载过程，您可以注册钩子过程来监督和更改其行为。要实现此目的，请使用 SysInit 单元中声明的 SetDliNotifyHook2 和 SetDliFailureHook2。另请参阅延迟加载 （Delphi） 中的代码示例。

动态加载

您可以通过直接调用 Windows API（包括 LoadLibrary、FreeLibrary 和 GetProcAddress）来访问库中的例程。它们也可以在 macOS，Linux 和 Android 上使用。这些函数在 Windows 的 Winapi.Windows.pas 单元和其他平台的 System.SysUtils.pas 中声明。在这种情况下，请使用过程类型变量来引用导入的例程。

例如：

```
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.
```

以这种方式导入例程时，在包含对 LoadLibrary 的调用的代码执行之前，不会加载库。该库稍后通过调用 FreeLibrary 卸载。这允许您节省内存并运行程序，即使它使用的某些库不存在。

编写动态库

注意：库在可以导出的内容方面比包受到的限制要大得多。库无法导出常量、类型和变量。也就是说，库中定义的类类型将不会在使用该库中的程序中看到。若要导出简单过程和函数以外的项，建议使用包。仅当需要与其他编程的互操作性时，才应考虑库。

在库中使用 Export 子句

动态可加载库的主源与程序的主源相同，只是它以保留字 library （而不是 program）开头。

只有库显式导出的例程才可供其他库或程序导入。下面的示例演示了一个库，其中包含两个导出的函数 Min 和 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.
```

如果您希望库可用于用其他语言编写的应用程序，最安全的方法是在导出函数的声明中指定 stdcall。其他语言可能不支持 Delphi 的默认寄存器调用约定。

库可以从多个单元构建。在这种情况下，库源文件经常简化为 use 子句、导出子句和初始化代码。例如：

```
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.
```

您可以将导出子句放在单元的接口或实现部分中。任何在其 use 子句中包含此类单元的库都会自动导出该单元的 exports 子句中列出的例程，而无需其自身的 export 子句。

当例程列在 export 子句中时，将导出该例程，其格式为：

```
exports entry1, ..., entryn;
```

其中，每个条目由过程、函数或变量的名称（必须在 exports 子句之前声明）组成，后跟参数列表（仅当导出重载的例程时）和可选的名称说明符。可以使用单元名称限定过程或函数名称。

（条目还可以包含指令 resident，为了向后兼容而维护该指令，编译器会忽略该指令。）

在 Win32 平台上，索引说明符由指令索引后跟一个介于 1 和 2,147,483,647 之间的数字常量组成。（对于更高效的程序，请使用低索引值。如果条目没有索引说明符，则在导出表中会自动为例程分配一个编号。）

注意：不鼓励使用仅支持向后兼容的索引说明符，否则可能会导致其他开发工具出现问题。

name 说明符由指令名称后跟字符串常量组成。如果条目没有名称说明符，则例程将以其

原始声明的名称导出，拼写和大小写相同。如果要以其他名称导出例程，请使用名称子句。例如：

```
exports
DoSomethingABC name 'DoSomething';
```

从可动态装入库导出重载函数或过程时，必须在 exports 子句中指定其参数列表。例如：

```
exports
Divide(X, Y: Integer) name 'Divide_Ints',
Divide(X, Y: Real) name 'Divide_Reals';
```

在 Win32 上，不要在重载例程的条目中包含索引说明符。

导出子句可以在程序或库的声明部分，或者在单元的接口或实现部分中的任何位置和任意次数出现。程序很少包含导出子句。

库初始化代码

库块中的语句构成库的初始化代码。每次加载库时，都会执行一次这些语句。它们通常执行注册窗口类和初始化变量等任务。库初始化代码还可以使用 DllProc 变量安装入口点过程。DllProc 变量类似于退出过程，退出过程中对此进行了描述;入口点过程在加载或卸载库时执行。

库初始化代码可以通过将 ExitCode 变量设置为非零值来指示错误。退出代码在系统单元中声明，默认为零，表示初始化成功。如果库的初始化代码将 ExitCode 设置为另一个值，则会卸载该库，并通知调用应用程序失败。同样，如果在执行初始化代码期间发生未经处理的异常，则会通知调用应用程序加载库失败。

下面是一个包含初始化代码和入口点过程的库示例：

```
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.
```

当库首次加载到内存中、线程启动或停止或卸载库时，将调用 DllProc。库使用的所有单元的初始化部分在库的初始化代码之前执行，这些单元的定版部分在库的入口点过程之后执行。

库中的全局变量

在共享库中声明的全局变量不能由 Delphi 应用程序导入。

一个库可以同时由多个应用程序使用，但每个应用程序在其自己的进程空间中都有一个库的副本，其中包含自己的一组全局变量。对于多个库（或一个库的多个实例）要共享内存，它们必须使用内存映射文件。有关详细信息，请参阅系统文档。

库和系统变量

系统单元中声明的几个变量对这些编程库特别感兴趣。使用 IsLibrary 确定代码是在应用程序中执行还是在库中执行;IsLibrary 在应用程序中始终为 False，在库中始终为 True。在库的生存期内，HInstance 包含其实例句柄。CmdLine 在库中始终为零。

DLLProc 变量允许库监视操作系统对库入口点进行的调用。此功能通常仅由支持多线程的库使用。DLLProc 用于多线程应用程序。对于所有退出行为，应使用完成部分，而不是退出过程。

若要监视操作系统调用，请创建采用单个整数参数的回调过程，例如：

```
procedure DLLHandler(Reason: Integer);
```

并将过程的地址分配给 DLLProc 变量。调用该过程时，它会将以下值之一传递给它。

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.

在过程的主体中，可以根据传递给过程的参数指定要执行的操作。

库中的异常和运行时错误

当引发异常但未在可动态加载的库中处理时，它会从库中传播到调用方。如果调用应用程序或库本身是用 Delphi 编写的，则可以通过正常的 try...except 语句来处理异常。

在 Win32 上，如果调用应用程序或库是用另一种语言编写的，则可以将异常作为异常代码为 $0EEDFADE 的操作系统异常进行处理。操作系统异常记录的 ExceptionInformation 数组中的第一个条目包含异常地址，第二个条目包含对 Delphi 异常对象的引用。

通常，不应让异常从库中逃逸。Delphi 异常映射到操作系统异常模型。

如果库不使用 SysUtils 单元，则会禁用异常支持。在这种情况下，当库中发生运行时错误时，调用应用程序将终止。由于库无法知道它是否是从 Delphi 程序调用的，因此它无法调用应用程序的退出过程;应用程序只是中止并从内存中删除。

共享内存管理器

在 Win32 上，如果 DLL 导出将长字符串或动态数组作为参数或函数结果传递的例程（无论是直接传递还是嵌套在记录或对象中），则 DLL 及其客户端应用程序（或 DLL）都必须使用 ShareMem 单元。如果一个应用程序或 DLL 使用 New 或 GetMem 分配内存，则通过调用另一个模块中的 Dispose 或 FreeMem 来释放内存，则情况也是如此。ShareMem 应始终是它发生的任何程序或库使用子句中列出的第一个单元。

ShareMem 是 BORLANDMM .DLL 内存管理器的接口单元，它允许模块共享动态分配的内存。BORLANDMM.DLL 必须与使用 ShareMem 的应用程序和 DLL 一起部署。当应用程序或 DLL 使用 ShareMem 时，其内存管理器将替换为 BORLANDMM.DLL 中的内存管理器。

包

包通常是导出项目的首选方法，而不是简单的过程和函数。仅当需要与其他编程的互操作性时，才应考虑库。

了解包

包是应用程序、IDE 或两者使用的专门编译的库。包允许您重新排列代码所在的位置，而不会影响源代码。这有时称为应用程序分区。

运行时包在用户运行应用程序时提供功能。设计时包用于在 IDE 中安装组件，并为自定义组件创建特殊的属性编辑器。单个包可以在设计时和运行时运行，并且设计时包通常通过在其 require 子句中引用运行时包来工作。

在 Win32 上，包文件以 .bpl（Borland 包库）扩展名结尾。

通常，包在应用程序启动时静态加载。但是，您可以使用 LoadPackage 和 UnloadPackage 例程（在 SysUtils 单元中）来动态加载包。

注意：当应用程序使用包时，每个打包单元的名称仍必须出现在引用它的任何源文件的 uses 子句中。

包声明和源文件

每个包都在单独的源文件中声明，该文件应以 .dpk 扩展名保存，以避免与包含 Delphi 代码的其他文件混淆。包源文件不包含类型、数据、过程或函数声明。相反，它包含：

- ✓ 包的名称。
- ✓ 新包所需的其他包的列表。这些是新包链接到的包。
- ✓ 编译包时包含或绑定到包中的单元文件的列表。包本质上是这些源代码单元的包装器，这些源代码单元提供已编译包的功能。

包声明的格式为：

```
package packageName;
requiresClause;
containsClause;
end.
```

其中 packageName 是任何有效的标识符。requiresClause 和 containsClause 都是可选的。例如，以下代码声明 DATAX 包：

```
package DATAX;
  requires
  rtl,
  contains Db, DBLocal, DBXpress, ... ;
end.
```

require 子句列出了所声明的包使用的其他外部包。它由指令 require 组成，后跟以逗号分隔的包名称列表，后跟分号。如果包不引用其他包，则不需要 require 子句。

contains 子句标识要编译并绑定到包中的单元文件。它由指令包含组成，后跟逗号分隔的单位名称列表，后跟分号。任何单元名称后都可以跟保留字 in 和源文件的名称，带或不带目录路径，用单引号引起来;目录路径可以是绝对路径，也可以是相对路径。例如：

```
contains MyUnit in 'C:\MyProject\MyUnit.pas';
```

注： 无法从使用该包的客户端访问打包单元中的线程局部变量（使用 threadvar 声明）。

命名包

编译的包涉及多个生成的文件。例如，名为 DATAX 的包的源文件是 DATAX.DPK，编译器从中生成可执行文件和二进制映像，称为

DATAX .BPL 和 DATAX.DCP

DATAX 用于在其他包的 require 子句中引用包，或者在应用程序中使用该包时引用包。包名称在项目中必须是唯一的。

requires 语句

require 子句列出了当前包使用的其他外部包。它的功能类似于单元文件中的 use 子句。require 子句中列出的外部包在编译时会自动链接到同时使用当前包和外部包中包含的单元之一的任何应用程序中。

如果包中包含的单元文件引用其他打包单元，则其他包应包含在第一个包的 require 子句中。如果 require 子句中省略了其他包，编译器将从其 .dcu 文件中加载引用的单元。

避免循环包引用

包的 require 子句中不能包含循环引用。这意味着
包不能在其自己的 require 子句中引用自身。
引用链必须终止，而不引用链中的任何包。如果包 A 需要包 B，则包 B 不能要求包 A; 如果包 A 需要包 B，包 B 需要包 C，则包 C 不能要求包 A。

重复的包引用

编译器忽略包的 require 子句中的重复引用。但是，为了编程的清晰度和可读性，应删除重复的引用。

contains 语句

包含子句标识要绑定到包中的单元文件。不要在包含子句中包含文件扩展名。

避免冗余源代码的使用

一个包不能列在另一个包的包含子句或单元的 use 子句中。
直接包含在包的 include 子句中的所有单元，或间接包含在这些单元的 uses 子句中的所有单元，在编译时都绑定到包中。包中包含的（直接或间接）单元不能包含在该包的要求子句中引用的任何其他包中。
一个单元不能（直接或间接）包含在同一个应用程序使用的多个包中。

编译包

包通常使用项目管理器生成的 .dpk 文件从 IDE 编译。您也可以直接从命令行编译 .dpk 文件。生成包含包的项目时，如有必要，将隐式重新编译包。

生成的文件

生成的文件下表列出了成功编译包后生成的文件。

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.

特定于包的编译器指令

下表列出了可以插入到源代码中的特定于包的编译器指令。

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.)

在源代码中包含 {$DENYPACKAGEUNIT ON} 可防止打包单元文件。包含 {$G-} 或 {$IMPORTEDDATA OFF} 可能会阻止一个包与其他包在同一应用程序中使用。

如果适用，其他编译器指令可以包含在包源代码中。

特定于包的命令行编译器开关

以下特定于包的开关可用于命令行编译器。

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.
-LEpath	Specifies the directory where the compiled package file .bpl will be placed.
-LNpath	Specifies the directory where the .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.

使用 -$G- 开关可能会阻止一个包与其他包在同一应用程序中使用。
如果适用，在编译包时可以使用其他命令行选项。

9. 接口

对象接口

对象接口或简称接口定义可由类实现的方法。接口声明为类，但不能直接实例化，也没有自己的方法定义。相反，支持接口的任何类都有责任为接口的方法提供实现。接口类型的变量可以引用其类实现该接口的对象;但是，只有接口中声明的方法才能使用此类变量调用。

接口提供了多重继承的一些优点，而没有语义上的困难。它们对于使用分布式对象模型（如 SOAP）也是必不可少的。使用分布式对象模型，支持接口的自定义对象可以与用 C++、Java 和其他语言编写的对象进行交互。

接口类型

接口（如类）只能在程序或单元的最外层范围内声明，而不能在过程或函数声明中声明。接口类型声明的格式为：

```
type interfaceName = interface (ancestorInterface) ['{GUID}'] memberList end;
```

警告：祖先接口和 GUID 规范是支持 Win32 COM 互操作性所必需的。如果要通过 COM 访问接口，请确保指定祖先接口和 GUID。

在大多数情况下，接口声明类似于类声明，但存在以下限制：

- ✓ 成员列表只能包含方法和属性。接口中不允许使用字段。
- ✓ 由于接口没有字段，因此属性读取和写入说明符必须是方法。
- ✓ 接口的所有成员都是公共的。不允许使用可见性说明符和存储说明符。（但数组属性可以声明为默认值。
- ✓ 接口没有构造函数或析构函数。它们不能实例化，除非通过实现其方法的类。
- ✓ 方法不能声明为虚拟、动态、抽象或重写。由于接口不实现自己的方法，因此这些指定没有意义。

下面是接口声明的示例：

```
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;
```

在某些接口声明中，接口保留字被替换为 dispinterface。

接口与继承

接口（如类）继承其所有祖先的方法。但是接口与类不同，它不实现方法。接口继承的是实现方法的义务，该义务传递给支持接口的任何类。

接口的声明可以指定祖先接口。如果未指定上级，则接口是 IInterface 的直接后代，IInterface 在系统单元中定义，并且是所有其他接口的最终上级。在 Win32 上，IInterface 声明了三种方法：QueryInterface，_AddRef 和_Release。

注意：IInterface 等同于 IUnknown。通常，对于独立于平台的应用程序，应使用 IInterface，并为包含 Win32 依赖项的特定程序保留使用 IUnknown。

查询接口提供了获取对对象支持的不同接口的引用的方法。_AddRef 和 _Release 为接口引用提供生存期内存管理。实现这些方法的最简单方法是从系统单元的 TInterfacedObject 派生实现类。也可以通过将任何这些方法实现为空函数来省去这些方法中的任何一个;但是，必须通过_AddRef 和_Release 来管理 COM 对象。

警告： 需要查询接口、_AddRef 和_Release 才能支持 Win32 COM 互操作性。如果要通过 COM 访问接口，请确保实现这些方法。

接口标识和 GUID

接口声明可以指定全局唯一标识符 （GUID），该标识符由紧靠成员列表前面的括号中的字符串文本表示。声明的 GUID 部分必须具有以下格式：

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

其中每个 x 是一个十六进制数字（0 到 9 或 A 到 F）。类型库编辑器会自动为新接口生成 GUID。还可以通过在代码编辑器中按 Ctrl+Shift+G 来生成 GUID。

GUID 是唯一标识接口的 16 字节二进制值。如果接口具有 GUID，则可以使用接口查询来获取对其实现的引用。

注意： GUID 仅用于 COM 互操作性。

在系统单元中声明的 TGUID 和 PGUID 类型用于操作 GUID。

```
type
    PGUID = ^TGUID;
    TGUID = packed record
      D1: Cardinal;
```

```
        D2: Word;
        D3: Word;
        D4: array[0..7] of Byte;
    end;
```

可以通过以下两种方式之一调用支持：

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

或者：

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

注意：SysUtils 单元提供了一个名为 Support 的重载函数，当类类型和实例支持 GUID 表示的特定接口时，该函数返回 true 或 false。支持函数以 Delphi 是和作为运算符的方式使用。显著区别在于，Support 函数可以将 GUID 或与 GUID 关联的接口类型作为正确的操作数，而 is 和 as 可以采用类型的名称。有关 is 和 as 的更多信息，请参见类引用。

接口的调用约定

接口方法的默认调用约定是寄存器，但是模块之间共享的接口（特别是如果它们是用不同的语言编写的）应该使用 stdcall 声明所有方法。在 Win32 上，可以使用安全调用来实现双接口的方法。

接口属性

接口中声明的属性只能通过接口类型的表达式访问;它们不能通过类类型变量访问。此外，接口属性仅在编译接口的程序中可见。

在接口中，属性读取和写入说明符必须是方法，因为字段不可用。

前向声明

以保留字接口和分号结尾且未指定祖先、GUID 或成员列表的接口声明是前向声明。前向声明必须由同一类型声明节中同一接口的定义声明解析。换句话说，在前向声明及其定义声明之间，除了其他类型的声明之外，什么都不会发生。

前向声明允许相互依赖的接口。例如：

```
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;
```

不允许使用相互派生的接口。例如，从 IControl 派生 IWindow 和从 IWindow 派生 IControl 是不合法的。

实现接口

声明接口后，必须先在类中实现该接口，然后才能使用它。类实现的接口在类的声明中指定，在类祖先的名称之后。

类实现声明

此类声明的形式为：

```
type className = class (ancestorClass, interface1, ..., interfaceN)
memberList
end;
```

例如：

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    // ...
  end;
```

声明一个名为 TMemoryManager 的类，该类实现 IMalloc 和 IErrorInfo 接口。当类实现接口时，它必须实现（或继承）接口中声明的每个方法的实现。

这是 System.TInterfacedObject（在 Windows 上）的声明。在其他平台上，声明略有不同）：

```
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 实现了 IInterface 接口。因此，TInterfacedObject 声明并实现了三个 IInterface 方法中的每一个。

实现接口的类也可以用作基类。（上面的第一个例子将 TMemoryManager 声明为 TInterfacedObject 的直接后代。每个接口都继承自 IInterface，实现接口的类必须实现 QueryInterface、_AddRef 和 _Release 方法。单元系统中的 TInterfacedObject 实现了这些方法，因此是一个方便的基础，从中派生实现接口的其他类。

实现接口时，其每个方法都映射到实现类中具有相同结果类型、相同调用约定、相同数量的参数以及每个位置相同类型参数的方法。默认情况下，每个接口方法都映射到实现类中同名的方法。

方法解析语句

可以通过在类声明中包含方法解析子句来重写默认的基于名称的映射。当类实现两个或多个具有相同名称方法的接口时，请使用方法解析子句来解决命名冲突。

方法解析子句的格式为：

```
procedure interface.interfaceMethod = implementingMethod;
```

或者：

```
function interface.interfaceMethod = implementingMethod;
```

其中实现方法是在类或其祖先之一中声明的方法。implementingMethod 可以是稍后在类声明中声明的方法，但不能是在另一个模块中声明的祖先类的私有方法。

例如，类声明：

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
   // ...
  end;
```

将 IMalloc 的 Alloc 和 Free 方法映射到 TMemoryManager 的 Allocate 和 Deallocate 方法。方法解析子句不能更改祖先类引入的映射。

更改继承的实现

后代类可以通过重写实现方法来更改特定接口方法的实现方式。这要求实现方法是虚拟的或动态的。

类还可以重新实现它从祖先类继承的整个接口。这涉及在后代类的声明中重新列出接口。例如：

```
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;
```

重新实现接口会隐藏同一接口的继承实现。因此，祖先类中的方法解析子句对重新实现的接口没有影响。

委托给接口类型属性

如果委托属性是接口类型，则该接口或其派生自的接口必须出现在声明该属性的类的祖先列表中。delegate 属性必须返回一个对象，该对象其类完全实现 implements 指令指定的接口，并且没有方法解析子句。例如：

```
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;
```

委托给类类型属性

如果委托属性是类类型，则在搜索封闭类及其祖先之前，将搜索该类及其祖先以实现指定接口的方法。因此，可以在属性指定的类中实现某些方法，并在声明属性的类中实现其他方法。方法解析子句可以按常规方式用于解决歧义或指定特定方法。一个接口不能由多个类类型属性实现。例如：

```
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;
```

接口引用

如果声明接口类型的变量，则该变量可以引用实现该接口的任何类的实例。

实现接口引用

接口引用变量允许您在编译时不知道接口的实现位置的情况下调用接口方法。但它们受以下约束：

- ✓ 接口类型表达式仅允许您访问接口中声明的方法和属性，而不能访问实现类的其他成员。
- ✓ 接口类型表达式不能引用其类实现后代接口的对象，除非该类（或它继承自的类）也显式实现祖先接口。

例如：

```
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;
```

在此示例中，A 声明为 IAncestor 类型的变量。由于 Tsomething 不会在其实现的接口中列出 IAncestor ，因此无法将 TSomething 实例分配给 A。但是，如果您将 TSomething 的声明更改为：

```
TSomething = class(TInterfacedObject, IAncestor, IDescendant)
  // ...
```

第一个错误将成为有效的赋值。D 声明为 IDescendant 类型的变量。虽然 D 引用了 TSomething 的实例，但你不能使用它来访问 TSomething 的 P2 方法，因为 P2 不是 IDescendant 的方法。但是，如果您将 D 的声明更改为：

```
D: TSomething;
```

第二个错误将成为有效的方法调用。

在 Win32 平台上，接口引用通常通过引用计数进行管理，这取决于从 System.IInterface 继承的_AddRef 和_Release 方法。使用引用计数的默认实现，当仅通过接口引用对象时，无需手动销毁它;当对对象的最后一个引用超出范围时，将自动销毁该对象。某些类实现接口以绕过此默认生存期管理，而某些混合对象仅在对象没有所有者时才使用引用计数。

全局接口类型变量只能初始化为 nil。

若要确定接口类型表达式是否引用对象，请将其传递给标准函数 Assigned 。

接口赋值兼容性

给定类类型的变量与该类实现的任何接口类型赋值兼容。接口类型的变量与任何祖先接口类型都与赋值兼容。值 nil 可以分配给任何接口类型变量。

可以将接口类型表达式赋值给变体。如果接口的类型是 IDispatch 或后代，则变体接收类型代码 varDispatch。否则，变体将接收类型代码 varUnknown。

类型代码为 varEmpty、varUnknown 或 varDispatch 的变体可以分配给 IInterface 变量。类型代码为 varEmpty 或 varDispatch 的变体可以分配给 IDispatch 变量。

接口类型转换

接口类型表达式可以强制转换为变体。如果接口的类型为 IDispatch 或后代，则生成的变体具有类型代码 varDispatch。否则，生成的变体具有类型代码 varUnknown。

类型代码为 varEmpty、varUnknown 或 varDispatch 的变体可以转换为 IInterface。类型代码为 varEmpty 或 varDispatch 的变体可以强制转换为 IDispatch。

接口查询

您可以使用 as 运算符执行已检查的接口类型转换。这称为接口查询，它根据对象的实际（运行时）类型从对象引用或其他接口引用生成接口类型表达式。接口查询的格式为：

```
object as interface
```

其中 Object 是接口或变体类型的表达式，或表示实现接口的类的实例，接口是使用 GUID 声明的任何接口。

如果对象为 nil，则接口查询返回 nil。否则，它将接口的 GUID 传递给对象中的 QueryInterface 方法，除非 QueryInterface 返回零，否则引发异常。如果 QueryInterface 返回零（指示对象的类实现接口），则接口查询返回对对象的接口引用。

强制转换接口引用为对象

as 运算符还可用于将接口引用转换回从中获取接口引用的对象。此强制转换仅适用于从 Delphi 对象获取的接口。例如：

```
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;
```

上面的示例演示如何获取从中获取接口引用的原始对象。当拥有接口引用根本不够时，此技术很有用。

如果未从给定类中提取接口，则 as 运算符将引发异常：

```
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;
```

还可以从对对象的接口引用执行普通类型转换（不安全）。与对象不安全强制转换的情况一样，此方法不会引发任何异常。不安全的对象到对象强制转换和不安全的接口到对象转换之间的区别在于，前者在不兼容类型的情况下返回有效指针，而后者返回 nil。该示例描述了不安全强制转换的使用：

```
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;
```

若要避免潜在的 nil 引用，请使用 is 运算符验证接口引用是否从给定类中提取：

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

注： 确保在使用不安全强制转换或 as 和 is 运算符时使用仅限 Delphi 的对象。

自动对象（win32)

其类实现 IDispatch 接口（在系统单元中声明）的对象是自动化对象。

使用变体访问自动化对象。当变体引用自动化对象时，您可以调用该对象的方法并通过该变体读取或写入其属性。为此，必须在其中一个单元或程序或库的使用子句中包含 ComObj。

调度接口类型

调度接口类型定义自动化对象通过 IDispatch 实现的方法和属性。对调度接口方法的调用在运行时通过 IDispatch 的调用方法路由;类不能实现调度接口。

调度接口类型声明的格式为：

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

其中

['{GUID}']

是可选的，接口包含属性和方法声明。调度接口声明类似于常规接口声明，但它们不能指定祖先。例如：

['{GUID}']

是可选的，接口包含属性和方法声明。调度接口声明类似于常规接口声明，但它们不能指定祖先。例如：

```
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;
```

调度接口方式

调度接口的方法是对底层 IDispatch 实现的 Invoke 方法的调用的原型。若要为方法指定自动化调度 ID，请在其声明中包含 dispid 指令，后跟整数常量;指定已使用的 ID 会导致错误。

在调度接口中声明的方法不能包含 dispid 以外的指令。参数和结果类型必须是可自动化的。换句话说，它们必须是 Byte、Currency、Real、Double、Longint、Integer、Single、Smallint、AnsiString、WideString、TDateTime、Variant、OleVariant、WordBool 或任何接口类型。

调度接口属性

调度接口的属性不包括访问说明符。它们可以声明为只读或只写。若要为属性指定调度 ID，请在其声明中包含 dispid 指令，后跟整数常量;指定已使用的 ID 会导致错误。数组属性可以声明为默认值。调度接口属性声明中不允许使用其他指令。

访问自动化对象

自动化对象方法调用在运行时绑定，不需要以前的方法声明。编译时不检查这些调用的有效性。

下面的示例演示自动化方法调用。CreateOleObject 函数（在 ComObj 中定义）返回对自动化对象的 IDispatch 引用，并且与变体 Word 的赋值兼容：

```
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;
```

可以将接口类型参数传递给自动化方法。

元素类型为 varByte 的变体数组是在自动化控制器和服务器之间传递二进制数据的首选方法。此类数组不需要转换其数据，并且可以使用 VarArrayLock 和 VarArrayUnlock 例程有效地访问。

自动化对象方法-调用语法

自动化对象方法调用或属性访问的语法类似于普通方法调用或属性访问的语法。但是，自动化方法调用可以同时使用位置参数和命名参数。（但某些自动化服务器不支持命名参数。

位置参数只是一个表达式。命名参数由参数标识符组成，后跟 ：= 符号，后跟表达式。位置参数必须位于方法调用中的任何命名参数之前。可以按任意顺序指定命名参数。

某些自动化服务器允许您从方法调用中省略参数，接受其默认值。例如：

```
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');
```

自动化方法调用参数可以是整数、实数、字符串、布尔值和变量类型。如果参数表达式仅包含变量引用，并且变量引用的类型为 Byte、Smallint、Integer、Single、Double、Currency、System.TDateTime、AnsiString、WordBool 或 Variant，则通过引用传递参数。如果表达式不属于这些类型之一，或者它不仅仅是变量，则按值传递参数。通过引用将参数传递给需要值参数的方法会导致 COM 从引用参数中提取值。将参数按值传递给需要引用参数的方法会导致错误。

双接口

双接口是通过自动化支持编译时绑定和运行时绑定的接口。双接口必须从 IDispatch 派生。

双接口的所有方法（从 IInterface 和 IDispatch 继承的方法除外）都必须使用 safecall 约定，并且所有方法参数和结果类型都必须是可自动化的。（可自动化的类型包括 Byte、Currency、Real、Double、Real48、Integer、Single、Smallint、AnsiString、ShortString、System/TDateTime、Variant、OleVariant 和 WordBool。

10. 内存管理

内存管理

本帮助主题描述在各种目标平台上使用的两个内存管理器，并简要描述变量的内存问题。

默认内存管理器

正在使用的内存管理器由应用程序的目标平台/编译器确定。
下表列出了每个平台的默认内存管理器。

Platform	Compiler	Memory Manager name
Win32	DCC32	FastMM (GETMEM.inc)
Win64	DCC64	FastMM (GETMEM.inc)
OSX64	DCCOSX64	Posix/64
OSXARM64	DCCOSXARM64	Posix/64
Linux64	DCCLINUX64	Posix/64
iOSDevice64	DCCIOSARM64	Posix/64
iOSSimARM64	DCCIOSSIMARM64	Posix/64
Android	DCCAARM	Posix/32
Android64	DCCAARM64	Posix/64

FastMM 内存管理器（Win32 和 Win64）

内存管理器管理应用程序中的所有动态内存分配和解除分配。New、Dispose、GetMem、ReallocMem 和 FreeMem 标准系统过程使用内存管理器。所有对象、动态数组和长字符串都通过内存管理器分配。

对于 Win32 和 Win64，默认的 FastMM 内存管理器针对分配大量中小型块的应用程序进行了优化，这是面向对象的应用程序和处理字符串数据的应用程序的典型特征。内存管理器针对单线程和多线程应用程序中的高效操作（高速和低内存开销）进行了优化。其他内存管理器（例如 Windows 中的 GlobalAlloc、LocalAlloc 和专用堆支持的实现）在这种情况下通常表现不佳，如果直接使用它们，则会降低应用程序的速度。

为了确保最佳性能，内存管理器直接与虚拟内存 API（VirtualAlloc 和 VirtualFree 函数）接口。

对于 Win32，内存管理器支持最大 2GB 的用户模式地址空间。

注：要将用户模式地址空间增加到 3GB，请参阅增加内存地址空间主题。

内存管理器块向上舍入为 4 字节的倍数，并包括一个 4 字节标头，其中存储块的大小和其他状态位。内存块的起始地址至少在 8 字节边界上对齐，或者可以选择在 16 字节边界上对齐，从而提高寻址时的性能。（参见 System.SetMinimumBlockAlign ）

对于 Win64，内存管理器在推测中支持高达 16EiB 的用户模式地址空间。

注意：实际的最大可分配大小取决于 CPU 实现和操作系统。例如，当前的 Intel/X64 实现支持高达 256TiB（48 位），Windows 7 Professional 支持高达 192GiB。

内存管理器块向上舍入为 16 字节的倍数，并包括一个 8 字节标头，其中存储块和其他状态位的大小。内存块的起始地址至少在 16 字节边界上对齐。

对于 Win32 和 Win64，内存管理器采用一种算法来预测未来的块重新分配，从而减少通常与此类操作相关的性能影响。重新分配算法还有助于减少地址空间碎片。内存管理器提供了一种不需要使用外部 DLL 的共享机制。

内存管理器包括报告功能，可帮助应用程序监视自己的内存使用情况和潜在的内存泄漏。

内存管理器提供了两个过程，即 GetMemoryManagerState 和 GetMemoryMap，它们允许应用程序检索内存管理器状态信息和内存使用情况的详细映射。

Posix 内存管理器（Posix 平台）

当目标平台/编译器为 macOS 64 位、macOS ARM 64 位、Linux 64 位、iOS 设备 64 位、iOS 模拟器 ARM 64 位、Android 32 位或 Android 64 位时，将使用 Posix 内存管理器。

所有内存管理函数/方法都使用 Posix 系统库，如下表所示：

RTL	POSIX function
AllocMem	calloc
FreeMem	free
GetMem	malloc
ReallocMem	realloc

变量

全局变量在应用程序数据段上分配，并在程序期间持续存在。局部变量（在过程和函数中声明）驻留在应用程序的堆栈中。每次调用过程或函数时，它都会分配一组局部变量;退出时，将释放局部变量。编译器优化可以更早地消除变量。

在 Win32 上，应用程序的堆栈由两个值定义：最小堆栈大小和最大堆栈大小。这些值通过 $MINSTACKSIZE 和 $MAXSTACKSIZE 编译器指令进行控制，默认分别为 16,384 （16K） 和 1,048,576 （1Mb）。保证应用程序具有可用的最小堆栈大小，并且不允许应用程序的堆栈增长到大于最大堆栈大小。如果没有足够的可用内存来满足应用程序的最低堆栈要求，Windows 将在尝试启动应用程序时报告错误。

如果 Win32 应用程序需要的堆栈空间多于最小堆栈大小指定的空间，则会以 4K 为增量自动分配额外的内存。如果额外堆栈空间分配失败，因为没有更多内存可用，或者因为堆栈的总大小将超过最大堆栈大小，则会引发 EStackOverflow 异常。（堆栈溢出检查是完全自动的。最初控制堆栈溢出检查的 $S 编译器指令被保留为向后兼容性。

使用 GetMem 或 New 过程创建的动态变量是堆分配的，并一直存在，直到使用 FreeMem 或 Dispose 解除分配它们。

长字符串、宽字符串、动态数组、变体和接口是堆分配的，但它们的内存是自动管理的。

内部储存格式

整数类型

整数值在 Delphi 中具有以下内部表示形式。

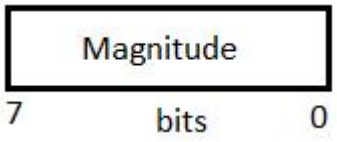
独立于平台的无符号整数类型

与平台无关的整数类型的值在任何平台上占用相同的位数。

无符号整数类型的值始终为正数，并且不像有符号整数类型那样涉及符号位。无符号整数类型的所有位都由值的大小占据，没有其他含义。

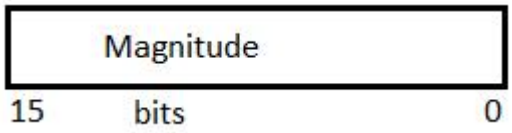
Byte,UInt8

Byte 和 UInt8 是 1 字节（8 位）无符号正整数。占用所有 8 位。



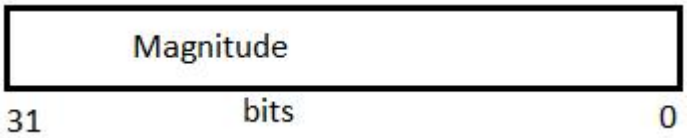
Word,UInt16

Word 和 UInt16 是 2 字节（16 位）无符号整数。



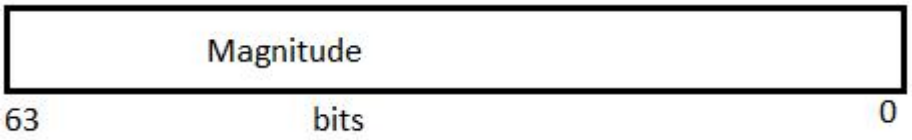
FixedUInt, Cardinal and UInt32

FixedUInt、Cardinal 和 UInt32 是 4 字节（32 位）无符号整数。



UInt64

UInt64 是 8 字节（64 位）无符号整数。



独立于平台的有符号整数类型

有符号整数类型的值表示一个数字的符号，由一个前导符号位表示，由最高有效位表示。正数的符号位为 0，负数的符号位为 1。正有符号整数中的其他位被幅度占用。在负有符号整数中，其他位被值大小（绝对值）的二进制补码表示占据。

要获得两者的补码到一个量级：

从右边开始，找到第一个“1”。

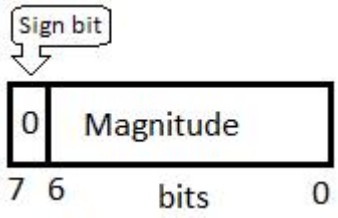
反转该位左侧的所有位。

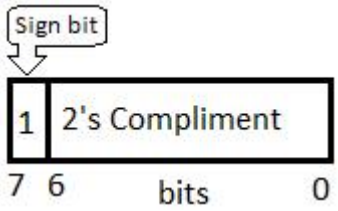
例如：

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

ShortInt, Int8

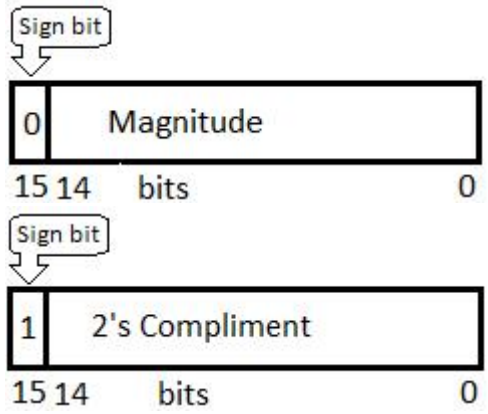
Shortint 和 Int8 是 1 字节（8 位）有符号整数。符号位占据最重要的第 7 位，幅度或 2 的补码占据其他 7 位。





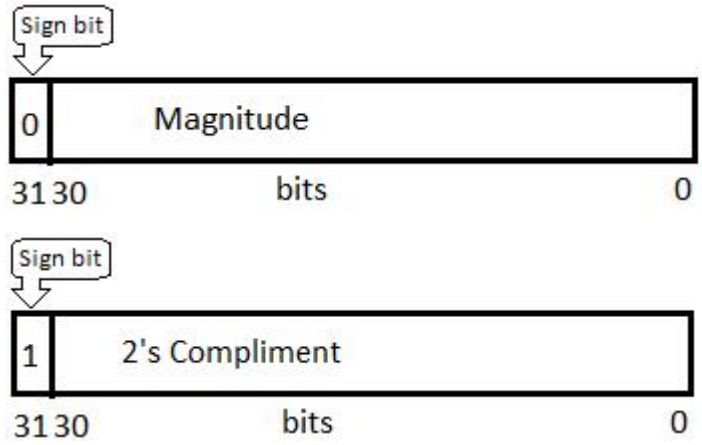
SmallInt 和 Int16

SmallInt 和 Int16 是 2 字节（16 位）有符号整数。



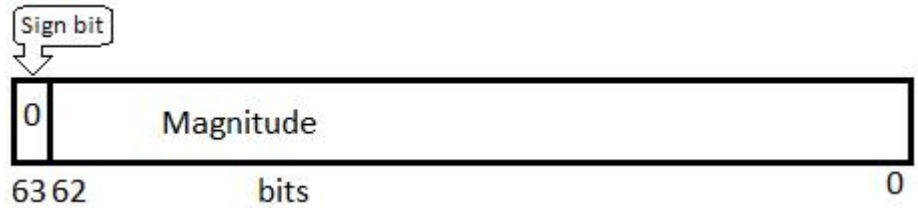
固定整数、整数和 Int32

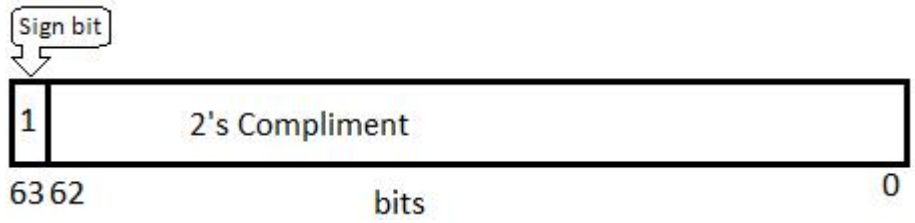
FixedInt、Integer 和 Int32 是 4 字节（32 位）有符号整数。



Int64

Int64 是 8 字节（64 位）有符号整数。





平台相关整数类型

转换与平台相关的整数类型以适合当前目标平台的位大小。在 64 位平台上，它们占用 64 位，在 32 位平台上，它们占用 32 位（LongInt 和 LongWord 类型除外）。当目标平台的大小与 CPU 平台的大小相同时，一个与平台相关的整数与 CPU 寄存器的大小完全匹配。当特定 CPU 类型和操作系统需要最佳性能时，通常使用这些类型。

无符号整数 NativeUInt

NativeUInt 是依赖于平台的无符号整数类型。NativeUInt 的大小和内部表示形式取决于当前平台。在 32 位平台上，NativeUInt 等效于 Cardinal 类型。在 64 位平台上，NativeUInt 等效于 UInt64 类型。

有符号整数 NativeInt

NativeInt 是依赖于平台的有符号整数类型。NativeInt 的大小和内部表示形式取决于当前平台。在 32 位平台上，NativeInt 等效于 Integer 类型。在 64 位平台上，NativeInt 等效于 Int64 类型。

LongInt 和 LongWord

LongInt 定义有符号整数类型，LongWord 定义无符号整数类型。LongInt 和 LongWord 平台相关的整数类型大小在每个平台上都会更改，但 64 位 Windows 保持不变（32 位）除外。

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

注意：RAD Studio 中的 32 位平台包括 32 位 Windows 和 32 位 Android。

在 Windows 以外的 64 位平台上，如果要使用：

- ✓ 32 位有符号整数类型，使用 Integer 或 FixedInt 而不是 LongInt。
- ✓ 32 位无符号整数类型，使用 Cardinal 或 FixedUInt 而不是 LongWord。

整数子范围类型

使用整数常量定义子范围类型的最小和最大边界时，将定义整数子范围类型。整数子范围类型表示整数类型（称为基类型）中的值的子集。基类型是包含指定范围（包含最小和最大边界）的最小整数类型。

整数子范围类型变量的内部数据格式取决于其最小和最大边界：

- ✓ 如果两个边界都在 -128..127 （ShortInt） 范围内，则变量将存储为有符号字节。
- ✓ 如果两个边界都在 0..255（Byte）范围内，则变量将存储为无符号字节。
- ✓ 如果两个边界都在 -32768..32767 （SmallInt） 范围内，则变量将存储为有符号字。
- ✓ 如果两个边界都在 0..65535 （Word） 范围内，则变量将存储为无符号字。
- ✓ 如果两个边界都在 -2147483648..2147483647 范围内（32 位平台和 64 位 Windows 平台上的 FixedInt 和 LongInt），则变量将存储为有符号双字。
- ✓ 如果两个边界都在 0..4294967295 范围内（在 32 位平台和 64 位 Windows 平台上为 FixedUInt 和 LongWord），则变量将存储为无符号双字。
- ✓ 如果两个边界都在 -2^63..2^63-1 范围内（64 位 iOS 平台上的 Int64 和 LongInt），则变量存储为有符号四重字。
- ✓ 如果两个边界都在 0..2^64-1 范围内（在 64 位 iOS 平台上为 UInt64 和 LongWord），则变量将存储为无符号四重字。

注意：“word”占用两个字节。

字符类型

在 32 位和 64 位平台上：

- ✓ Char 和 WideChar 存储为无符号字变量，通常使用 UTF-16 或 Unicode 编码。
- ✓ AnsiChar 类型存储为无符号字节。在 Delphi 2007 年及更早的版本中，Char 被代表为 AnsiChar。与短字符串一起使用的字符类型始终是 AnsiChar，并存储在无符号字节值中。
- ✓ 默认的长字符串类型（字符串）现在是 UnicodeString，它的引用计数类似于 AnsiString，以前的默认长字符串类型。与旧代码的兼容性可能需要使用 AnsiString 类型。
- ✓ WideString 由 UnicodeString 等 WideChar 组成，但不计算引用。

布尔类型

布尔类型存储为 Byte，ByteBool 存储为 Byte，WordBool 类型存储为 Word，LongBool 存储为 Longint。

布尔值可以假定值为 0（假）和 1（真）。ByteBool、WordBool 和 LongBool 类型可以假定值为 0（False）或非零 （True）。

枚举类型

如果枚举的值不超过 256 个，并且类型声明为 {$Z 1} 状态（默认值），则枚举类型将存储为无符号字节。如果枚举类型的值超过 256 个，或者该类型是在 {$Z 2} 状态下声明的，则它存储为无符号字。如果枚举类型声明为 {$Z 4} 状态，则将其存储为无符号双字。

实数类型

实数类型存储符号（+或-）、指数和有效数的二进制表示形式。实值的形式为

+/- significand * 2^exponent

其中，有效数在二进制小数点左侧有一个位（即 0 <= 有效数 < 2）。

在下面的图像中，最高有效位始终位于左侧，最低有效位始终位于右侧。顶部的数字表示每个字段的宽度（以位为单位），最左侧的项目存储在最高地址。例如，对于 Real48 值，e 存储在第一个字节中，f 存储在接下来的五个字节中，s 存储在最后一个字节的最高有效位中。

Real48 类型

在 32 位和 64 位平台上，6 字节（48 位）Real48 数字分为三个字段。

1	39	8
s	f	e

如果 0 < e <= 255，则数字的值 v 由下式给出：

v = (-1)s * 2(e-129) * (1.f)

如果 e = 0，则 v = 0。

Real48 类型不能存储非正态、NaN 和无穷大 （Inf）。当存储在 Real48 中时，非正常值变为零，而如果尝试将它们存储在 Real48 中，则 NaN 和无穷大会产生溢出错误。

Single 类型

在 32 位和 64 位平台上，4 字节（32 位）单个数字分为三个字段。

1	8	23
s	e	f

数字的值 v 由下式给出：

- ✓ 如果 0 < e < 255，则 v = （-1）s * 2（e-127） * （1.f）
- ✓ 如果 e = 0 且 f <> 0，则 v = （-1）s * 2（-126） * （0.f）
- ✓ 如果 e = 0 且 f = 0，则 v = （-1）s * 0

- ✓ 如果 e = 255 且 f = 0，则 v = （-1）s * Inf
- ✓ 如果 e = 255 且 f <> 0，则 v 是 NaN

Double 类型

在当前实现中，Real 类型等效于 Double。

在 32 位和 64 位平台上，8 字节（64 位）双精度数分为三个字段。

1	11	52
s	e	f

数字的值 v 由下式给出：

如果 0 < e < 2047，则 v = （-1）s * 2（e-1023） * （1.f）
如果 e = 0 且 f <> 0，则 v = （-1）s * 2（-1022） * （0.f）
如果 e = 0 且 f = 0，则 v = （-1）s * 0
如果 e = 2047 且 f = 0，则 v = （-1）s * Inf
如果 e = 2047 且 f <> 0，则 v 是 NaN

Extended 类型

扩展版在 32 位 Intel 平台上提供比其他实际类型更高的精度，但可移植性较差。如果要创建要跨平台共享的数据文件，请谨慎使用扩展。请注意：

在 32 位 Windows Intel 平台上，扩展数字表示为 10 字节（80 位）。扩展号码分为四个字段。

1	15	1	63
s	e	i	f

数字的值 v 由下式给出：

- ✓ 如果 0 <= e < 32767，则 v = （-1）s * 2（e-16383） * （i.f）
- ✓ 如果 e = 32767 且 f = 0，则 v = （-1）s * Inf
- ✓ 如果 e = 32767 且 f <> 0，则 v 是 NaN

在 Windows（64 位 Linux 或 64 位 macOS Intel）以外的 64 位 Intel 平台上，扩展类型为 16 字节，即使数据位仅使用 10 个字节。

但是，在 64 位 Windows Intel 平台和 ARM 平台（64 位 macOS ARM、31 位 Android、64 位 Android、64 位 iOS）上，扩展类型是 Double 的别名，只有 8 个字节。这种差异会对浮点运算中的数值精度产生负面影响。有关详细信息，请参阅多设备应用程序的 Delphi 注意事项。

Comp 类型

8 字节（64 位）Comp 编号存储为有符号的 64 位整数。

Currency 类型

8 字节（64 位）货币编号存储为缩放和有符号的 64 位整数，其中 4 个最低有效数字隐式表示 4 位小数。

指针类型

在 32 位平台上，指针类型以 4 个字节的形式存储为 32 位地址。
在 64 位平台上，指针类型以 8 个字节的形式存储为 64 位地址。
指针值 nil 存储为零。

短字符串类型

短字符串字符串占用的字节数与其最大长度加 1 的字节数一样多。第一个字节包含字符串的当前动态长度，后续字节包含字符串的字符。
长度字节和字符被视为无符号值。最大字符串长度为 255 个字符加上一个长度字节（字符串 [255]）。

长字符串类型

UnicodeString 或 AnsiString 类型的字符串变量在 32 位平台上占用 4 个字节的内存（在 64 位平台上占用 8 个字节），这些平台包含指向动态分配的字符串的指针。当字符串变量为空（包含零长度字符串）时，字符串指针为 nil，并且没有动态内存与字符串变量关联。对于非空字符串值，字符串指针指向动态分配的内存块，该内存块除了包含描述字符串的信息外，还包含字符串值。下表显示了长字符串内存块的布局。
UnicodeString 数据类型的格式（32 位和 64 位）

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

“偏移量”行中的数字显示字符串指针中描述字符串内容的字段偏移量，字符串指针指向字符串数据字段（偏移量 = 0），其中包含包含实际字符串值的内存块。
字符串内存块末尾的 NULL 字符由编译器和内置字符串处理例程自动维护。这样就可以将字符串直接类型转换为以 null 结尾的字符串。
另请参阅“新字符串类型：UnicodeString”。

对于字符串文本，编译器生成一个内存块，其布局与动态分配的字符串相同，但引用计数为 -1。字符串常量的处理方式相同，与文字的唯一区别是它们是指向 -1 引用计数器块的指针。

当指向字符串结构（源）的指针分配给字符串变量（目标）时，引用计数器指示如何完成此操作。通常，目标的引用计数会减少，而源的引用计数会增加，因为指针（源和目标）在分配后都将指向相同的内存块。

如果源引用计数为 -1（字符串常量），则会创建一个引用计数为 1 的新结构。如果目标不为 nil，则引用计数器将减小。如果达到 0，则从内存中释放结构。如果目标为 nil，则不会对其执行其他操作。然后，目标将指向新结构。

```
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 1destination := 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.
```

如果源引用计数不是 -1，则它递增，目标将指向它。

```
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.
```

注意：任何字符串变量都不能指向引用计数为 0 的结构。结构在达到 0 引用计数时始终被释放，当它们的引用计数为 -1 时，无法修改。

宽字符串类型

在 32 位平台上，宽字符串变量占用 4 个字节的内存（在 64 位上占用 8 个字节），其中包含指向动态分配的字符串的指针。当宽字符串变量为空（包含零长度字符串）时，字符串指针为 nil，并且没有动态内存与字符串变量关联。对于非空字符串值，字符串指针指向动态分配的内存块，该内存块除了包含 32 位长度指示器外还包含字符串值。下表显示了 Windows 上宽字符串内存块的布局。

宽字符串动态内存布局（32 位和 64 位）

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

字符串长度是字节数，因此它是字符串中包含的宽字符数的两倍。

宽字符串内存块末尾的 NULL 字符由编译器和内置字符串处理例程自动维护。这样就可以将宽字符串直接类型转换为以 null 结尾的字符串。

集合类型

集合是一个位数组，其中每个位指示元素是否在集合中。集合中的最大元素数为 256，因此集合占用的字节永远不会超过 32 个字节。特定集占用的字节数等于

(Max div 8) - (Min div 8) + 1

其中 Max 和 Min 是集合的基本类型的上限和下限。特定元素 E 的字节号为

(E div 8) - (Min div 8)

并且该字节内的位号为

E mod 8

其中 E 表示元素的序号值。如果可能，编译器将集合存储在 CPU 寄存器中，但如果集合大于与平台相关的整数类型，或者程序包含获取集合地址的代码，则该集合始终驻留在内存中。

静态数组类型

静态数组变量占用的内存由 Length（array） * SizeOf（array[Low（array）] ） 以字节为单位的计算来定义。静态数组变量完全作为父数据结构的一部分进行分配。静态数组存储为数组组件类型的连续元素序列。具有最低索引的组件存储在最低的内存地址。存储一个多维数组，最右边的维度首先增加。

动态数组类型

在 32 位平台上，动态数组变量占用 4 个字节的内存（64 位上占用 8 个字节），其中包含指向动态分配数组的指针。当变量为空（未初始化）或保存零长度数组时，指针为 nil，并且没有动态内存与该变量关联。对于非空数组，变量指向动态分配的内存块，该内存块除了包含 32 位（在 Win64 上为 64 位）长度指示器和 32 位引用计数之外，还包含该数组。下表显示了动态阵列内存块的布局。

动态阵列内存布局（32 位和 64 位）

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

记录类型

当记录类型声明为 {$A+} 状态（默认值）时，并且声明不包含打包修饰符时，该类型为解压缩记录类型，并且记录的字段将对齐以便 CPU 根据平台进行高效访问。对齐方式由每个字段的类型控制。每种数据类型都有一个固有的对齐方式，该对齐方式由编译器自动计算。对齐方式可以是 1、2、4 或 8，表示字节边界，为了提供最有效的访问，必须存储该类型的值。下表列出了所有数据类型的对齐方式。

类型对齐掩码（仅限 32 位）

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

为了确保解压缩记录类型中的字段正确对齐，如果需要，编译器会在对齐方式为 2 的字段之前插入一个未使用的字节，在对齐方式为 4 的字段之前最多插入 3 个未使用的字节。最后，编译器将记录的总大小向上舍入到由任何字段的最大对齐方式指定的字节边界。

使用通用类型规范隐式打包字段

Delphi 编译器的早期版本（如 Delphi 7 和更早版本）将压缩对齐方式隐式应用于一起声明的字段，即具有通用类型规范的字段。如果指定指令 {$OLDTYPELAYOUT ON}，较新的编译器可以重现该行为。此指令对具有通用类型规范的字段进行字节对齐（打包），即使声明不包括 pack 修饰符并且记录类型未声明为 {$A-} 状态也是如此。

因此，例如，给定以下声明：

```
{$OLDTYPELAYOUT ON}
type
  TMyRecord = record
```

```
    A, B: Extended;
    C: Extended;
  end;
{$OLDTYPELAYOUT OFF}
```

A 和 B 被打包（在字节边界上对齐），因为指定了 {$OLDTYPELAYOUT ON} 指令，并且因为 A 和 B 共享相同的类型规范。但是，对于单独声明的 C 字段，编译器使用默认行为并使用未使用的字节填充结构，以确保该字段显示在四字边界上。

当记录类型声明为 {$A-} 状态时，或者当声明包含打包修饰符时，记录的字段不会对齐，而是分配连续偏移量。这种打包记录的总大小只是所有字段的大小。由于数据对齐方式可能会更改，因此最好将要写入磁盘的任何记录结构打包到磁盘或将内存传递到使用不同版本的编译器编译的另一个模块。

文件类型

文件类型表示为记录。类型化文件和非类型化文件在 32 位平台上占用 592 字节，在 64 位平台上占用 616 字节，布局如下：

```
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;
```

文本文件在 Win 32 上占用 730 字节，在 Win64 上占用 754 字节，布局如下：

```
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;
```

句柄包含文件的句柄（打开文件时）。
“模式”字段可以采用以下值之一：

```
const
    fmClosed = $D7B0;
    fmInput= $D7B1;
    fmOutput = $D7B2;
    fmInOut= $D7B3;
```

其中 fmClosed 表示文件已关闭，fmInput 和 fmOutput 表示已重置 （fmInput） 或重写（fmOutput） 的文本文件，fmInOut 表示已重置或重写的类型化或非类型化文件。任何其他值都指示未分配文件变量（因此未初始化）。

“用户数据”字段可用于用户编写的例程来存储数据。

Name 包含文件名，文件名是以空字符 结尾的字符序列。

对于类型化文件和非类型化文件，RecSize 包含以字节为单位的记录长度，并且“专用”字段未使用但保留。

对于文本文件，BufPtr 是指向 BufSize 字节缓冲区的指针，BufPos 是缓冲区中要读取或写入的下一个字符的索引，BufEnd 是缓冲区中有效字符的计数。OpenFunc、InOutFunc、FlushFunc 和 CloseFunc 是指向控制文件的 I/O 例程的指针;请参阅设备功能。标志确定换行符样式，如下所示。

bit 0 clear	LF line breaks
bit 0 set	CRLF line breaks

所有其他标志位都保留供将来使用。

注意：对于使用 UnicodeString 类型（默认的 Delphi 字符串类型），类单元中的各种流

类型（TFileStream、TStreamReader、TStreamWriter 等）更有用，因为较旧的文件类型具有有限的 Unicode 功能，尤其是旧的文本文件类型。

过程类型

在 32 位平台上，过程指针存储为指向过程或函数入口点的 32 位指针。方法指针存储为指向方法入口点的 32 位指针，后跟指向对象的 32 位指针。

在 64 位平台上，过程指针存储为指向过程或函数入口点的 64 位指针。方法指针存储为指向方法入口点的 64 位指针，后跟指向对象的 64 位指针。

类类型

在 32 位平台（Win32 和 Android）上，类类型值存储为指向类实例的 32 位指针（以及 64 位平台上的 64 位指针），称为对象。对象的内部数据格式类似于记录的内部数据格式。对象的字段按声明顺序存储为一系列连续变量。字段始终对齐，对应于解压缩的记录类型。因此，对齐方式对应于对象中字段的最大对齐方式。从祖先类继承的任何字段都存储在后代类中定义的新字段之前。

在 32 位平台上，每个对象的第一个 4 字节字段（64 位平台上的第一个 8 字节字段）是指向类的虚拟方法表 （VMT） 的指针。每个类只有一个 VMT（不是每个对象一个）;不同的类类型（无论多么相似）从不共享 VMT。VMT 由编译器自动构建，从不由程序直接操作。指向 VMT 的指针（由构造函数方法自动存储在它们创建的对象中）也永远不会由程序直接操作。

VMT 的布局如下表所示。在 32 位平台上，在正偏移量处，VMT 由 32 位方法指针（64 位平台上的 64 位方法指针）列表组成 - 类类型中每个用户定义的虚拟方法一个 - 按声明顺序排列。每个插槽都包含虚拟方法的相应入口点的地址。此布局与 C++对向表和 COM 兼容。在负偏移量处，VMT 包含许多德尔福实现内部的字段。应用程序应使用 TObject 中定义的方法查询此信息，因为布局在 Delphi 语言的未来实现中可能会发生变化。

虚拟方法表布局

Offset Win32, macOS	Offset Win64	Offset iOS/ARM, Android/ARM	Type	Description	Constant in System.pas
-88	-200	-108	Pointer	Pointer to virtual method table (or nil)	vmtSelfPtr
-84	-192	-104	Pointer	Pointer to interface table (or nil)	vmtIntfTable
-80	-184	-100	Pointer	Pointer to Automation information table (or nil)	vmtAutoTable
-76	-176	-96	Pointer	Pointer to instance initialization table (or nil)	vmtInitTable
-72	-168	-92	Pointer	Pointer to type information table (or nil)	vmtTypeInfo

-68	-160	-88	Pointer	Pointer to field definition table (or nil)	vmtFieldTable
-64	-152	-84	Pointer	Pointer to method definition table (or nil)	vmtMethodTable
-60	-144	-80	Pointer	Pointer to dynamic method table (or nil)	vmtDynamicTable
-56	-136	-76	Pointer	Pointer to short string containing class name	vmtClassName
-52	-128	-72	Cardinal	Instance size in bytes	vmtInstanceSize
-48	-120	-68	Pointer	Pointer to a pointer to ancestor class (or nil)	vmtParent
n/a	n/a	-64	Pointer	Entry point of __ObjAddRef method	vmtObjAddRef
n/a	n/a	-60	Pointer	Entry point of __ObjRelease method	vmtObjRelease
-44	-112	-56	Pointer	Entry point of Equals method	vmtEquals
-40	-104	-52	Pointer	Entry point of GetHashCode method	vmtGetHashCode
-36	-96	-48	Pointer	Entry point of ToString method	vmtToString
-32	-88	-44	Pointer	Pointer to entry point of SafecallException method (or nil)	vmtSafeCallException
-28	-80	-40	Pointer	Entry point of AfterConstruction method	vmtAfterConstruction
-24	-72	-36	Pointer	Entry point of BeforeDestruction method	vmtBeforeDestruction
-20	-64	-32	Pointer	Entry point of Dispatch method	vmtDispatch
-16	-56	-28	Pointer	Entry point of DefaultHandler method	vmtDefaultHandler
-12	-48	-24	Pointer	Entry point of NewInstance method	vmtNewInstance
-8	-40	-20	Pointer	Entry point of FreeInstance method	vmtFreeInstance
-4	-32	-16	Pointer	Entry point of Destroy destructor	vmtDestroy
0	0	0	Pointer	Entry point of first	

				user-defined virtual method	
4	8	4	Pointer	Entry point of second user-defined virtual method	

类引用类型

在 32 位平台（Win32 和 Android）上，类引用值存储为指向类的虚拟方法表 （VMT） 的 32 位指针。

在 64 位平台（Win64、64 位 Linux、64 位 iOS 和 64 位 Android）上，类引用值存储为指向类的虚拟方法表 （VMT） 的 64 位指针。

变体类型

变体依赖于将数据装箱和拆箱到对象包装器中，以及 Delphi 帮助程序类来实现与变体相关的 RTL 函数。

在 32 位平台上，变体存储为 16 字节记录，其中包含类型代码和代码给定类型的值（或对值的引用）。在 64 位平台上，变体存储为 24 字节记录。“系统”和“系统变体”单元定义变体的常量和类型。

TVarData 类型表示变量的内部结构（在 Windows 上，这与 COM 和 Win32 API 使用的变量类型相同）。TVarData 类型可用于变量的类型转换，以访问变量的内部结构。TVarData 记录包含以下字段：

- ✓ TVarType 类型的 VType 字段具有字（16 位）大小。VType 包含低 12 位变体的类型代码（由 varTypeMask = $FFF 常量定义的位）。此外，varArray = $2000 位可以设置为指示变体是一个数组，并且可以将 varByRef （= $4000） 位设置为指示变体包含引用而不是值。
- ✓ “保留 1”、“保留 2”和“保留 3（字大小）”字段未使用。

TVarData 记录的剩余 8 个字节（32 位平台）或 16 个字节（64 位平台）的内容取决于 VType 字段，如下所示：

- ✓ 如果既未设置 varArray 也未设置 varByRef 位，则变量包含给定类型的值。
- ✓ 如果设置了 varArray 位，则变体包含指向定义数组的 TVarArray 结构的指针。每个数组元素的类型由 VType 字段中的 varTypeMask 位给出。
- ✓ 如果设置了 varByRef 位，则变量包含对 VType 字段中的 varTypeMask 和 varArray 位给出的类型值的引用。

varString 类型代码是私有的。包含 varString 值的变体永远不应传递给非 Delphi 函数。在 Windows 平台上，Delphi 的自动化支持会自动将 varString 变体转换为 varOleStr 变体，然后再将它们作为参数传递给外部函数。

11. 程序控制

在执行应用程序项目之前，了解传递参数和函数结果处理的概念非常重要 参数和函数结果的处理由多个因素决定，包括调用约定、参数语义以及要传递的值的类型和大小。

参数传递

参数通过 CPU 寄存器或堆栈传输到过程和函数，具体取决于例程的调用约定。有关调用约定的信息，请参阅有关调用约定的主题。

按值与按引用

变量 （var） 参数始终通过引用传递，作为指向实际存储位置的 32 位指针。
值和常量 （const） 参数按值或引用传递，具体取决于参数的类型和大小：

- ✓ 序号参数作为 8 位、16 位、32 位或 64 位值传递，使用的格式与相应类型的变量相同。
- ✓ 始终在堆栈上传递一个实参数。单个参数占用 4 个字节，双精度、比较或货币参数占用 8 个字节。Real48 占用 8 个字节，Real48 值存储在较低的 6 个字节中。扩展值占用 12 个字节，扩展值存储在较低的 10 个字节中。
- ✓ 短字符串参数作为指向短字符串的 32 位指针传递。
- ✓ 长字符串或动态数组参数作为 32 位指针传递到为长字符串分配的动态内存块。为空长字符串传递值 nil。
- ✓ 指针、类、类引用或过程指针参数作为 32 位指针传递。
- ✓ 方法指针作为两个 32 位指针在堆栈上传递。实例指针被推到方法指针之前，以便方法指针占据最低的地址。
- ✓ 在 register 和 pascal 约定下，变量参数作为 32 位指针传递到 Variant 值。
- ✓ 1、2 或 4 字节的集合、记录和静态数组作为 8 位、16 位和 32 位值传递。较大的集合、记录和静态数组作为指向值的 32 位指针传递。此规则的一个例外是，记录始终在 cdecl、stdcall 和安全调用约定下直接在堆栈上传递;以这种方式传递的记录的大小向上舍入到最近的双字边界。
- ✓ 开放数组参数作为两个 32 位值传递。第一个值是指向数组数据的指针，第二个值比数组中的元素数少 1。

当堆栈上传递两个参数时，每个参数占用 4 个字节的倍数（整数个双字）。对于 8 位或 16 位参数，即使参数仅占用一个字节或一个字，也会作为双精度字传递。双字未使用部分的内容未定义。

Pascal、cdecl、stdcall 和 safecall 约定

在 pascal、cdecl、stdcall 和 safecall 约定下，所有参数都在堆栈上传递。根据 pascal 约定，参数按其声明的顺序（从左到右）推送，因此第一个参数最终位于最高地址，最后一个参

数最终位于最低地址。在 cdecl、stdcall 和 safecall 约定下，参数按声明的相反顺序（从右到左）推送，因此第一个参数最终位于最低地址，最后一个参数最终位于最高地址。

寄存器约定

根据寄存器约定，最多在 CPU 寄存器中传递三个参数，其余参数（如果有）在堆栈上传递。参数按声明顺序传递（与帕斯卡约定一样），符合条件的前三个参数按该顺序在 EAX、EDX 和 ECX 寄存器中传递。实数、方法指针、变体、Int64 和结构化类型不符合寄存器参数的条件，但所有其他参数都符合。如果三个以上的参数符合寄存器参数的条件，则前三个参数在 EAX、EDX 和 ECX 中传递，其余参数按声明顺序推送到堆栈上。例如，给定声明：

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

对 Test 的调用将 EAX 中的 A 作为 32 位整数传递，在 EDX 中将 B 作为指向 Char 的指针，在 ECX 中将 D 作为指向长字符串内存块的指针传递;C 和 E 按该顺序作为两个双字和一个 32 位指针推送到堆栈上。

寄存器保存约定

过程和函数必须保留 EBX、ESI、EDI 和 EBP 寄存器，但可以修改 EAX、EDX 和 ECX 寄存器。在汇编程序中实现构造函数或析构函数时，请务必保留 DL 寄存器。调用过程和函数时，假设 CPU 的方向标志已清除（对应于 CLD 指令），并且必须在清除方向标志的情况下返回。

注意：Delphi 语言过程和函数通常假设 FPU 堆栈为空：编译器在生成代码时尝试使用所有八个 FPU 堆栈条目。

使用 MMX 和 XMM 指令时，请确保保留 xmm 和 mm 寄存器的值。调用 Delphi 函数时，假设 x87 FPU 数据寄存器可供 x87 浮点指令使用。也就是说，编译器假定 EMMS/FEMMS 指令已在 MMX 操作之后调用。德尔菲函数不对 xmm 寄存器的状态和内容做出任何假设。它们不保证 xmm 寄存器的内容保持不变。

处理函数结果

以下约定用于返回函数结果值。

- ✓ 如果可能，序号结果将在 CPU 寄存器中返回。字节在 AL 中返回，单词在 AX 中返回，双字在 EAX 中返回。
- ✓ 实际结果在浮点协处理器的堆栈顶部寄存器 （ST（0）） 中返回。对于 Currency 类型的函数结果，ST（0）中的值按 10000 缩放。例如，货币值 1.234 在 ST（0）中返回为 12340。
- ✓ 对于字符串、动态数组、方法指针或变量结果，效果与将函数结果声明为声明参数之后的附加 var 参数相同。换句话说，调用方传递一个额外的 32 位指针，该指针指向要在其中返回函数结果的变量。
- ✓ Int64 在 EDX：EAX 中返回。
- ✓ 指针、类、类引用和过程指针结果在 EAX 中返回。

- ✓ 对于静态数组、记录和设置结果，如果值占用一个字节，则以 AL 形式返回;如果该值占用两个字节，则以 AX 形式返回;如果该值占用四个字节，则在 EAX 中返回。否则，结果将在声明的参数之后传递给函数的附加 var 参数中返回。

处理方法调用

方法使用与普通过程和函数相同的调用约定，不同之处在于每个方法都有一个额外的隐式参数 Self，该参数是对调用该方法的实例或类的引用。Self 参数作为 32 位指针传递。

- ✓ 根据寄存器约定，Self 的行为就像在所有其他参数之前声明一样。因此，它始终在 EAX 寄存器中传递。
- ✓ 根据 pascal 约定，Self 的行为就像是在所有其他参数之后声明的（包括有时为函数结果传递的附加 var 参数）。因此，它被推到最后，最终的地址低于所有其他参数。
- ✓ 在 cdecl、stdcall 和 safecall 约定下，Self 的行为就像在所有其他参数之前声明一样，但在为函数结果传递的附加 var 参数（如果有）之后。因此，它是最后一个被推送的，除了额外的 var 参数。

构造函数和析构函数使用与其他方法相同的调用约定，只是传递了一个额外的布尔标志参数来指示构造函数或析构函数调用的上下文。

构造函数调用的标志参数中的值 False 指示构造函数是通过实例对象或使用继承的关键字调用的。在这种情况下，构造函数的行为类似于普通方法。构造函数调用的标志参数中的值 True 指示构造函数是通过类引用调用的。在这种情况下，构造函数创建 Self 给出的类的实例，并返回对 EAX 中新创建的对象的引用。

析构函数调用的标志参数中的值 False 表示析构函数是使用继承的关键字调用的。在这种情况下，析构函数的行为类似于普通方法。析构函数调用的标志参数中的值 True 指示析构函数是通过实例对象调用的。在这种情况下，析构函数在返回之前解除分配 Self 给出的实例。

标志参数的行为就像在所有其他参数之前声明一样。根据寄存器惯例，它在 DL 寄存器中传递。根据帕斯卡约定，它被推到所有其他参数之前。在 cdecl、stdcall 和 safecall 约定下，它被推送在 Self 参数之前。

由于 DL 寄存器指示构造函数还是析构函数是调用堆栈中最外层的，因此必须在退出之前还原 DL 的值，以便可以正确调用 BeforeDestruction 或 AfterConstruction。

理解程序退出

退出过程可确保在程序终止之前执行特定操作，例如更新和关闭文件。ExitProc 指针变量允许您安装退出过程，以便始终在程序终止过程中调用它，无论终止是正常的、由对 Stop 的调用强制的还是运行时错误的结果。退出过程不带任何参数。

注： 建议您对所有退出行为使用完成部分而不是退出过程。退出过程仅适用于可执行文件。为。DLL （Win32） 您可以使用类似的变量 DllProc，该变量在加载和卸载库时调用。对于包，必须在完成部分中实现退出行为。所有退出过程在执行最终确定部分之前调用。

单元和程序可以安装退出程序。单元可以安装退出过程作为其初始化代码的一部分，依靠该过程关闭文件或执行其他清理任务。

如果实施得当，退出程序是退出程序链的一部分。这些过程以与安装相反的顺序执行，确保一个单元的退出代码不会先于依赖于它的任何单元的退出代码执行。要保持链完整，您必须先保存 ExitProc 的当前内容，然后再将其指向您自己的退出过程的地址。此外，退出过程中的第一个语句必须重新安装 ExitProc 的保存值。

以下代码显示了退出过程的框架实现：

```
var
ExitSave: Pointer;

procedure MyExit;

begin
    ExitProc := ExitSave; // always restore old vector first
 .
    .
    .
end;

begin
    ExitSave := ExitProc;
    ExitProc := @MyExit;
    .
    .
    .
end.
```

在进入时，代码将 ExitProc 的内容保存在 ExitSave 中，然后安装 MyExit 过程。当作为终止过程的一部分调用时，MyExit 做的第一件事是重新安装以前的退出过程。

运行时库中的终止例程不断调用退出过程，直到 ExitProc 变为 nil。为了避免无限循环，ExitProc 在每次调用之前都设置为 nil，因此仅当当前退出过程为 ExitProc 分配地址时，才会调用下一个退出过程。如果在退出过程中发生错误，则不会再次调用它。

退出过程可以通过检查 ExitCode 整数变量和 ErrorAddr 指针变量来了解终止的原因。在正常终止的情况下，ExitCode 为零，ErrorAddr 为 nil。如果通过调用 Stop 终止，ExitCode 包含传递给 Halt 的值，并且 ErrorAddr 为 nil。如果由于运行时错误而终止，ExitCode 包含错误代码，ErrorAddr 包含无效语句的地址。

最后一个退出过程(运行时库安装的过程)关闭输入和输出文件。如果 ErrorAddr 不为 nil，则输出运行时错误消息。若要输出自己的运行时错误消息，请安装一个退出过程，该过程检查 ErrorAddr 并在不为 nil 时输出一条消息;在返回之前，请将 ErrorAddr 设置为 nil，以便其他退出过程不会再次报告错误。

一旦运行时库调用了所有退出过程，它就会返回到操作系统，并将存储在 ExitCode 中的值作为返回代码传递。

12. X86 内联汇编

注意：内联汇编代码在 Win32、Win64 和 macOS 的 x86 平台上受支持，但 iOS 设备和 Android 设备的 Delphi 编译器不支持。

使用内联汇编代码

内置汇编程序允许您在 Delphi 程序中编写汇编代码。它具有以下功能：

- ✓ 允许内联汇编。
- ✓ 支持英特尔奔腾 4、英特尔 MMX 扩展、SIMD 流指令扩展 （SSE） 和 AMD 速龙（包括 3D Now！
- ✓ 支持英特尔 64 架构，但有一些限制。
- ✓ 允许在汇编语句中使用 Delphi 标识符，例如常量、类型和变量。
- ✓ 不提供宏支持，但允许纯汇编函数或过程。

作为内置汇编程序的替代方法，可以链接到包含外部过程和函数的对象文件。有关详细信息，请参阅主题外部声明。如果您有要在应用程序中使用的外部汇编代码，则应考虑使用 Delphi 语言重写它，或者使用内联汇编器最低限度地重新实现它。

内联汇编程序可用于：

DCC32.EXE，Delphi 命令行编译器
DCC64.EXE，Delphi 64 位命令行编译器

但是，适用于 iOS 设备和 Android 设备的 Delphi 编译器不支持内联程序集。

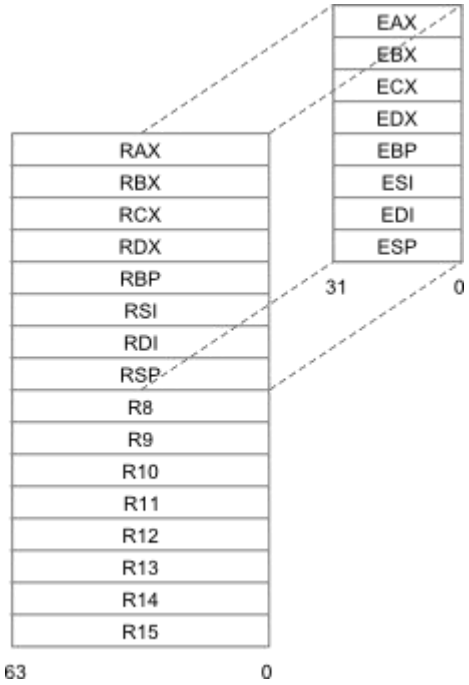
使用 asm 语句

内置汇编程序可通过 asm 语句访问，其形式为：

```
asm statementList end
```

其中 statementList 是由分号、行尾字符或 Delphi 注释分隔的一系列汇编语句。
asm 语句中的注释必须采用 Delphi 样式。分号并不表示该行的其余部分是注释。
保留字内联和指令汇编程序仅用于向后兼容。它们对编译器没有影响。

使用寄存器



32 位

通常，asm 语句中的寄存器使用规则与外部过程或函数的规则相同。asm 语句必须保留 EDI、ESI、ESP、EBP 和 EBX 寄存器，但可以自由修改 EAX、ECX 和 EDX 寄存器。在进入 asm 语句时，EBP 指向当前堆栈帧，ESP 指向堆栈顶部。除了 ESP 和 EBP 之外，asm 语句在进入语句时不能假定寄存器内容。

64 位

与 x64 应用程序二进制接口 （ABI） 一致，必须在内联程序集函数中保留和恢复以下寄存器的内容：R12、R13、R14、R15、RDI、RSI、RBX、RBP、RSP、XMM4、XMM5、XMM6、XMM7、XMM8、XMM8、XMM9、XMM10、XMM11、XMM12、XMM13、XMM14 和 XMM15。

内联汇编函数的前四个参数分别通过 RCX、RDX、R8 和 R9 传递，但使用 XMMO、XMM1、XMM2、XMM3 的浮点参数除外。数学协处理器通常不从 x64 代码中使用。用于功能参数的寄存器可以自由修改。

对跨平台代码使用条件编译

对于具有内联程序集代码的现有函数，必须使用条件定义来区分平台。函数应该在平台之间具有通用的功能原型。例：

```
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}
```

不带$ELSE 的示例：

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

有关预定义条件的详细信息，请参阅条件编译。预定义的条件。

汇编语法

语句

汇编语句的语法为：

```
Label: Prefix Opcode Operand1, Operand2
```

其中 Label 是标签，Prefix 是汇编前缀操作码（操作码），Opcode 是汇编指令操作码或指令，操作数是汇编表达式。标签和前缀是可选的。有些操作码只采用一个操作数，有些则不采用任何操作数。

允许在汇编语句之间添加注释，但不允许在其中使用注释。例如：

```
MOV AX,1 {Initial value}   { OK }
MOV CX,100 {Count}             { OK }
MOV {Initial value} AX,1; { Error! }
MOV CX, {Count} 100            { Error! }
```

标签

标签在内置程序集语句中使用，就像在 Delphi 语言中一样，方法是在语句之前编写标签和冒号。标签的长度没有限制。与 Delphi 一样，标签必须在包含 asm 语句的块的标签声明部分中声明。此规则的一个例外是本地标签。

本地标签是以符号 （@） 开头的标签。它们由一个符号后跟一个或多个字母、数字、下划线或符号组成。本地标签的使用仅限于 asm 语句，本地标签的范围从 asm 保留字扩展到包含它的 asm 语句的末尾。不必声明本地标签。

指令操作码

内置汇编程序支持所有英特尔记录的操作码，供一般应用使用。请注意，可能不支持操作系统特权指令。具体而言，支持以下指令系列：

IA-32
- Pentium family
- Pentium Pro and Pentium II
- Pentium III
- Pentium 4

Intel 64

此外，内置汇编器支持以下指令集扩展

Intel SSE (including SSE4.2)
AMD 3DNow! (from the AMD K6 onwards)
AMD Enhanced 3DNow! (from the AMD Athlon onwards)

有关每个指令的完整说明，请参阅微处理器文档。

自动跳转大小

除非另有说明，否则内置汇编程序通过自动选择最短，因此最有效的跳转指令形式来优化跳转指令。这种自动跳转大小调整适用于无条件跳转指令 （JMP），以及当目标是标签（而不是过程或函数）时的所有条件跳转指令。

对于无条件跳转指令 （JMP），如果到目标标签的距离为 -128 到 127 字节，则内置汇编器会生成一个短跳转（一个字节的操作码后跟一个字节位移）。否则，它会生成一个近跳转（一个字节的操作码后跟一个两个字节的位移）。

对于条件跳转指令，如果到目标标签的距离为 -128 到 127 字节，则生成短跳转（一个字节操作码后跟一个字节位移）。否则，内置汇编程序会生成到目标标签的近跳转。

跳转到过程和函数的入口点总是近在咫尺。

指令

内置汇编器支持三个汇编定义指令：DB（定义字节）、DW（定义字）和 DD（定义双字）。每个都生成与指令后面的逗号分隔操作数相对应的数据。

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. 定义字节：生成字节序列。每个操作数可以是值介于 128 和 255 之间的常量表达式，也可以是任意长度的字符串。常量表达式生成一个字节的代码，字符串生成一个字节序列，其值对应于每个字符的 ASCII 代码
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. 定义单词：生成一系列单词。每个操作数可以是值介于 32,768 和 65,535 之间的常量表达式，也可以是地址表达式。对于地址表达式，内置汇编程序会生成一个近指针，即包含地址偏移部分的单词。
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. 定义双字：生成一系列双字。每个操作数可以是值介于 2,147,483,648 和 4,294,967,295 之间的常量表达式，也可以是地址表达式。对于地址表达式，内置汇编程序会生成一个远指针，一个包含地址偏移部分的单词，后跟一个包含地址段部分的单词。
DQ	Define quad word: defines a quad word for Int64 values. 定义四字：定义 Int64 值的四字。

DB、DW 和 DD 指令生成的数据始终存储在代码段中，就像其他内置程序集语句生成的代码一样。要在数据段中生成未初始化或初始化的数据，应使用 Delphi var 或 const 声明。

DB、DW 和 DD 指令的一些示例如下：

```
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;
```

当标识符位于 DB、DW 或 DD 指令之前时，它会导致在指令的位置声明字节、字或双字大小的变量。例如，汇编程序允许执行以下操作：

```
ByteVar           DB    ?
WordVar            DW    ?
IntVar            DD    ?
// …
MOV       AL,ByteVar
MOV       BX,WordVar
MOV ECX,IntVar
```

内置汇编程序不支持此类变量声明。唯一可以在内联程序集语句中定义的符号是标签。所有变量必须使用 Delphi 语法声明;前面的构造可以替换为：

```
var
  ByteVar: Byte;
  WordVar: Word;
  IntVar: Integer;
// …
asm
  MOV AL,ByteVar
  MOV BX,WordVar
  MOV ECX,IntVar
end;
```

可以使用小和大来确定位移的宽度：

```
MOV EAX, [LARGE $1234]
```

此指令生成具有 32 位位移（$00001234）的“正常”移动：

```
MOV EAX, [SMALL $1234]
```

第二条指令将生成一个具有地址大小覆盖前缀和 16 位位移（1234 美元）的移动。
SMALL 可用于节省空间。以下示例生成地址大小覆盖和 2 字节地址（总共三个字节）：

```
MOV EAX, [SMALL 123]
```

而不是：

```
MOV EAX, [123]
```

这将不生成地址大小覆盖和 4 字节地址（总共 4 个字节）。
另外两个指令允许程序集代码访问动态和虚拟方法：VMTOFFSET 和 DMTINDEX。
VMTOFFSET 从虚拟方法表 （VMT） 的开头检索虚拟方法参数的虚拟方法指针表条目的偏移量（以字节为单位）。此指令需要一个完全指定的类名，并将方法名称作为参数（例如，TExample.VirtualMethod），或者需要接口名称和接口方法名称。
DMTINDEX 检索传递的动态方法的动态方法表索引。此指令还需要一个完全指定的类名，并将方法名作为参数，例如 TExample.DynamicMethod。若要调用动态方法，请使用包含从 DMTINDEX 获取的值的 （E）SI 寄存器调用 System.@CallDynaInst。
注意：带有消息指令的方法作为动态方法实现，也可以使用 DMTINDEX 技术调用。例如：

```
TMyClass = class
   procedure x; message MYMESSAGE;
end;
```

以下示例使用 DMTINDEX 和 VMTOFFSET 来访问动态和虚拟方法：

```
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.
```

操作数

内联汇编操作数是由常量、寄存器、符号和运算符组成的表达式。
在操作数中，以下保留字具有预定义的含义：

内置汇编程序保留字

中央处理器寄存器

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, ...

数据和运算符

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

保留字始终优先于用户定义的标识符。例如：

```
var
   Ch: Char;// …
asm
   MOV    CH, 1
end;
```

将 1 加载到 CH 寄存器中，而不是加载到 Ch 变量中。要访问与保留字同名的用户定义符号，必须使用与号 （&） 覆盖运算符：

```
MOV &Ch, 1
```

最好避免使用与内置汇编程序保留字同名的用户定义标识符。

汇编表达式

内置汇编程序将所有表达式计算为 32 位整数值。它不支持浮点和字符串值，字符串常量除外。

表达式由表达式元素和运算符生成，每个表达式都有一个关联的表达式类和表达式类型。

Delphi 表达式和汇编表达式的区别

Delphi 表达式和内置汇编表达式之间最重要的区别在于汇编表达式必须解析为常量值。换句话说，它必须解析为可在编译时计算的值。例如，给定声明：

```
const
   X = 10;
```

```
  Y = 20;
var
  Z: Integer;
```

以下是有效声明：

```
asm
  MOV       Z,X+Y
end;
```

因为 X 和 Y 都是常量，所以表达式 X + Y 是编写常量 30 的便捷方法，生成的指令只是将值 30 移动到变量 Z 中。但如果 X 和 Y 是变量：

```
var
  X, Y: Integer;
```

内置汇编器无法在编译时计算 X + Y 的值。在这种情况下，要将 X 和 Y 的总和移动到 Z 中，您可以使用：

```
asm
  MOV          EAX,X
  ADD          EAX,Y
  MOV          Z,EAX
end;
```

在 Delphi 表达式中，变量引用表示变量的内容。但在汇编表达式中，变量引用表示变量的地址。在 Delphi 中，表达式 X + 4（其中 X 是变量）表示 X 加 4 的内容，而对于内置汇编器，它表示地址比 X 地址高四个字节的单词内容。因此，即使您被允许编写：

```
asm
  MOV          EAX,X+4
end;
```

此代码不会将 X 加 4 的值加载到 AX 中;相反，它加载存储在 X 之外四个字节的单词的值。在 X 的内容中添加 4 的正确方法是：

```
asm
  MOV          EAX,X
  ADD          EAX,4
end;
```

表达式元素

表达式的元素是常量、寄存器和符号。

数值常量

数值常量必须是整数，其值必须介于 2,147,483,648 和 4,294,967,295 之间。

默认情况下，数字常量使用十进制表示法，但内置汇编程序还支持二进制、八进制和十六进制。二进制表示法通过在数字后写 B 来选择，八进制表示法通过在数字后写 O 来选择，十六进制表示法通过在数字后写 H 或在数字前写 $ 来选择。

数字常量必须以数字 0 到 9 或 $ 字符之一开头。使用 H 后缀编写十六进制常量时，如果第一个有效数字是数字 A 到 F 之一，则数字前面需要一个额外的零。例如，0BAD4H 和 $BAD 4 是十六进制常量，但 BAD4H 是标识符，因为它以字母开头。

字符串常量

字符串常量必须括在单引号或双引号中。与括起来的引号类型相同的两个连续引号仅计为一个字符。下面是字符串常量的一些示例：

```
'Z'
'Delphi'
'Windows'
"That's all folks"
'"That''s all folks," he said.'
'100'
''''
""""
```

DB 指令中允许使用任意长度的字符串常量，并导致分配包含字符串中字符的 ASCII 值的字节序列。在所有其他情况下，字符串常量不能超过四个字符，并表示可以参与表达式的数值。字符串常量的数值计算公式为：

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

其中 Ch1 是最右边（最后一个）字符，Ch4 是最左边（第一个）字符。如果字符串短于四个字符，则假定最左侧的字符为零。下表显示了字符串常量及其数值。

字符串示例及其值：

String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H

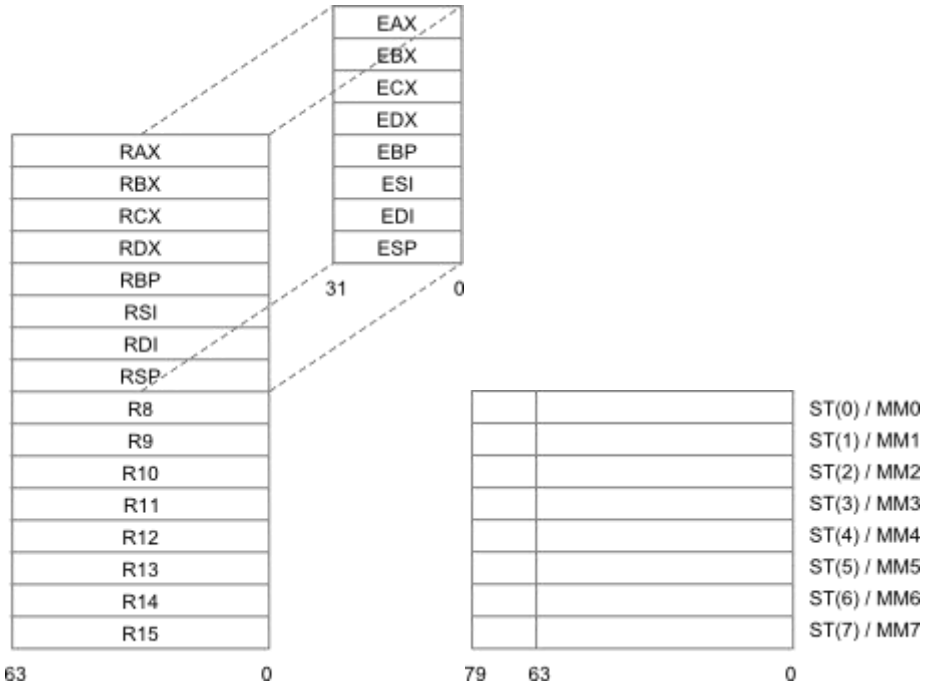
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

寄存器

以下保留符号表示内联汇编程序中的 CPU 寄存器：
中央处理器寄存器

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 通用寄存器、x86 FPU 数据寄存器和 x64 SSE 数据寄存器



XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15

127 0

当操作数仅由寄存器名称组成时，它称为寄存器操作数。所有寄存器都可以用作寄存器操作数，某些寄存器可以在其他上下文中使用。

基本寄存器（BX 和 BP）和索引寄存器（SI 和 DI）可以写在方括号内以指示索引。有效的基本/索引寄存器组合为 [BX]、[BP]、[SI]、[DI]、[BX+SI]、[BX+DI]、[BP+SI] 和 [BP+DI]。您还可以使用所有 32 位寄存器进行索引，例如 [EAX+ECX]、[ESP] 和 [ESP+EAX+5]。

支持段寄存器（ES、CS、SS、DS、FS 和 GS），但在 32 位应用程序中通常没有用。

符号 ST 表示 8087 浮点寄存器堆栈上最顶层的寄存器。八个浮点寄存器中的每一个都可以使用 ST（X） 引用，其中 X 是介于 0 和 7 之间的常数，指示与寄存器堆栈顶部的距离。

符号

内置汇编程序允许您访问汇编语言表达式中的几乎所有 Delphi 标识符，包括常量、类型、变量、过程和函数。此外，内置汇编程序实现了特殊符号@Result，它对应于函数主体中的 Result 变量。例如，函数：

```
function Sum(X, Y: Integer): Integer;
begin
  Result := X + Y;
end;
```

可以用汇编语言写成：

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    ADD     EAX,Y
    MOV     @Result,EAX
  end;
end;
```

以下符号不能在 asm 语句中使用：

- ✓ 标准过程和函数（例如，Writeln 和 Chr）。
- ✓ 字符串、浮点和设置常量（加载寄存器时除外）。
- ✓ 未在当前块中声明的标签。
- ✓ 函数外部的@Result 符号。

下表总结了可在 asm 语句中使用的符号类型。
内置汇编程序识别的符号：

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

禁用优化后，局部变量（在过程和函数中声明的变量）始终在堆栈上分配并相对于 EBP 进行访问，局部变量符号的值是其与 EBP 的有符号偏移量。汇编程序会自动在对局部变量的引用中添加 [EBP]。例如，给定声明：

```
var Count: Integer;
```

在函数或过程中，指令：

```
MOV       EAX,Count
```

组装成 MOV EAX,[EBP4].

内置汇编器将 var 参数视为 32 位指针，var 参数的大小始终为 4。访问 var 参数的语法与访问值参数的语法不同。若要访问 var 参数的内容，必须先加载 32 位指针，然后访问它指向的位置。例如：

```
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;
```

标识符可以在 asm 语句中限定。例如，给定声明：

```
type
   TPoint = record
      X, Y: Integer;
   end;
   TRect = record
      A, B: TPoint;
   end;var
   P: TPoint;
   R: TRect;
```

可以在 ASM 语句中使用以下结构来访问字段：

```
MOV      EAX,P.X
MOV      EDX,P.Y
MOV      ECX,R.A.X
MOV      EBX,R.B.Y
```

类型标识符可用于动态构造变量。以下每个指令都生成相同的机器代码，该机器代码将 [EDX] 的内容加载到 EAX 中。

```
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
```

表达式分类

内置汇编程序将表达式分为三类：寄存器、内存引用和即时值。

仅由寄存器名称组成的表达式是寄存器表达式。寄存器表达式的示例包括 AX、CL、DI 和 ES。寄存器表达式用作操作数，指示汇编程序生成在 CPU 寄存器上运行的指令。

表示内存位置的表达式是内存引用。Delphi 的标签、变量、类型常量、过程和函数都属于这一类。

不是寄存器且不与内存位置关联的表达式是即时值。此组包括 Delphi 的非类型化常量和类型标识符。

即时值和内存引用在用作操作数时会导致生成不同的代码。例如：

```
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;
```

由于 Start 是即时值，因此第一个 MOV 被组装成移动即时指令。但是，第二个 MOV 被转换为移动内存指令，因为 Count 是内存参考。在第三个 MOV 中，括号将 Start 转换为内存引用（在本例中为数据段中偏移量为 10 处的字）。在第四个 MOV 中，OFFSET 运算符将 Count 转换为即时值（数据段中 Count 的偏移量）。

括号和 OFFSET 运算符相辅相成。以下 asm 语句生成与前一个 asm 语句的前两行相同的机器代码：

```
asm
   MOV         EAX,OFFSET [Start]
   MOV         EBX,[OFFSET Count]
end;
```

内存参考和即时值进一步分类为可重定位或绝对值。重定位是链接器为符号分配绝对地址的过程。可重定位表达式表示需要在链接时重定位的值，而绝对表达式表示不需要此类重定位的值。通常，引用标签、变量、过程或函数的表达式是可重定位的，因为这些符号的最终地址在编译时是未知的。仅对常量进行操作的表达式是绝对的。

内置汇编程序允许您对绝对值执行任何操作，但它将对可重定位值的操作限制为常量的加法和减法。

表达式类型

每个内置汇编程序表达式都有一个类型，或者更准确地说是一个大小，因为汇编程序将表达式的类型简单地视为其内存位置的大小。例如，Integer 变量的类型为 4，因为它占用 4 个字节。内置汇编程序尽可能执行类型检查，因此在说明中：

```
var
   QuitFlag: Boolean;
   OutBufPtr: Word;// …
asm
   MOV         AL,QuitFlag
   MOV         BX,OutBufPtr
end;
```

汇编程序检查 QuitFlag 的大小是否为 1（一个字节），OutBufPtr 的大小是否为 2（一个

单词）。说明：

```
MOV         DL,OutBufPtr
```

产生错误，因为 DL 是一个字节大小的寄存器，而 OutBufPtr 是一个字。内存引用的类型可以通过类型转换更改;这些是编写上一条指令的正确方法：

```
MOV         DL,BYTE PTR OutBufPtr
MOV         DL,Byte(OutBufPtr)
MOV         DL,OutBufPtr.Byte
```

这些 MOV 指令都引用 OutBufPtr 变量的第一个（最不重要）字节。
在某些情况下，内存引用是非类型的。一个例子是括在方括号中的即时值 （Buffer）：

```
procedure Example(var Buffer);
asm
    MOV AL,      [Buffer]
    MOV CX,      [Buffer]
    MOV EDX,     [Buffer]
end;
```

内置汇编程序允许这些指令，因为表达式 [Buffer] 没有类型。[缓冲区] 表示“缓冲区指示的位置的内容”，类型可以从第一个操作数（AL 的字节、CX 的 word 和 EDX 的双字符）确定。
如果无法从另一个操作数确定类型，则内置汇编程序需要显式类型转换。例如：

```
INC       BYTE PTR [ECX]
IMUL      WORD PTR [EDX]
```

下表总结了内置汇编程序提供的预定义类型符号以及当前声明的任何 Delphi 类型。
预定义类型符号：

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

表达式运算符

内置汇编程序提供了多种运算符。优先规则与德尔菲语言的规则不同;例如，在 ASM 语句中，AND 的优先级低于加法和减法运算符。下表按优先级降序列出了内置汇编程序的表达

式运算符。

内置汇编程序表达式运算符的优先级

Operators	Remarks	Precedence
&		highest
(...), [...],., HIGH, LOW		
+, -	unary + and -	
:		
OFFSET, TYPE, PTR, *, /, MOD, SHL, SHR, +, -	binary + and -	
NOT, AND, OR, XOR		lowest

下表定义了内置汇编程序的表达式运算符：

内置汇编表达式运算符的定义：

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.

汇编过程和函数

您可以使用内联汇编语言代码编写完整的过程和函数，而无需包含 begin...end 语句。

编译器优化

您可以编写的函数类型的示例如下：

```
function LongMul(X, Y: Integer): Longint;
asm
   MOV    EAX,X
   IMUL       Y
end;
```

编译器对这些例程执行多项优化：

- ✓ 不会生成任何代码来将值参数复制到局部变量中。这会影响所有字符串类型的值参数以及大小不是 1、2 或 4 字节的其他值参数。在例程中，必须将此类参数视为 var 参数。
- ✓ 除非函数返回字符串、变体或接口引用，否则编译器不会分配函数结果变量;对@Result 符号的引用是错误的。对于字符串、变体和接口，调用方始终分配一个@Result 指针。
- ✓ 编译器仅为嵌套例程、具有本地参数的例程或堆栈上具有参数的例程生成堆栈帧。
- ✓ 局部变量是局部变量的大小，参数是参数的大小。如果局部变量和参数都为零，则没有入场代码，退出代码仅由 RET 指令组成。

自动生成的例程进入和退出代码如下所示：

```
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
```

如果局部变量包含变体、长字符串或接口，则它们将初始化为零，但不会最终确定。

函数结果

汇编语言函数返回其结果，如下所示。

32 位

序号值以 AL（8 位值）、AX（16 位值）或 EAX（32 位值）返回。
实值以 ST（0） 形式返回协处理器寄存器堆栈上的值。（货币值按 10000 缩放。
指针（包括长字符串）在 EAX 中返回。
短字符串和变体在 @Result 指向的临时位置返回。

64 位

在 RAX 中返回大小为 8 字节或更小的值。
实值以 XMM0 格式返回。
其他类型由指针值驻留在 RAX 中的引用返回，其内存由调用例程分配。

英特尔 64 细节（伪操作）

x64 函数必须完全用汇编或 Pascal 编写，即不支持汇编语句，仅支持内联汇编函数。
提供了伪操作来帮助管理 x64 中的堆栈使用：.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.

PC 映射异常的堆栈展开

Mac 和 Linux 系统使用程序计数器 （PC） 映射的异常，而 Windows （Win32） 使用堆栈上链接的注册记录。

展开程序集例程程序集例程无法捕获异常。此外，在调用程序集例程时必须小心，因为展开细节。如果非汇编代码调用汇编代码，而汇编代码又调用非汇编代码，则此方面很重要。引发/抛出异常时，堆栈开卷器会丢弃异常通过的堆栈帧。开卷过程必须随时恢复各种关键寄存器。其中包括堆栈指针和帧指针。你可以在汇编代码中做一些事情来使这个展开过程变得不可能，你必须避免做这些事情。一个关键项目是，如果按可变量修改堆栈指针，则开卷器无法在没有帮助的情况下展开帧。例如：

```
sub esp， eax
```
或

```
sub ESP， 0xF
And esp，0xFFFFFFF0
```

汇编代码中的上述任一构造都使开卷器无法展开给定的过程。为了使开卷器能够展开包含如下指令的过程，必须在代码中包含堆栈帧：

```
push EBP
MOV EBP， ESP
```

堆栈帧的创建必须是代码的前两条指令。编译汇编代码时，编译器会识别此指令序列，编译器会为例程生成专门的展开信息。如果帧不存在，编译器将检查整个程序集例程，尝试分析代码如何修改堆栈，以便在引发/引发异常时展开修改。

13. 泛型

提供泛型概述、术语列表、泛型语法更改摘要，以及有关声明和使用参数化类型、指定泛型约束和使用重载的详细信息。

概述

泛型的工作原理

术语泛型或泛型类型描述平台中可按类型参数化的事物集。术语泛型可以指泛型类型或泛型方法，即泛型过程和泛型函数。

泛型是一组抽象工具，允许将算法（如过程或函数）或数据结构（如类、接口或记录）与算法或数据结构使用的一个或多个特定类型分离。

通过用类型参数替换一个或多个特定类型，可以使在其定义中使用其他类型的方法或数据类型更加通用。然后，将这些类型参数添加到方法或数据结构声明中的类型参数列表中。这类似于通过用参数名称替换文本常量主体中的实例并将参数添加到过程的参数列表中来使过程更通用的方式。

例如，维护对象列表（TObject 类型）的 TMyList 类可以通过将 TObject 的使用替换为类型参数名称（如“T”）并将类型参数添加到类的类型参数列表中以使其成为 TMyList，使其更具可重用性和类型安全<T>性。

泛型类型或方法的特定用途（实例化）可以通过在使用点向泛型类型或方法提供类型参数来实现。提供类型参数的操作通过将泛型定义中类型参数的实例替换为相应的类型参数来有效地构造新的类型或方法。

例如，该列表可能用作 TMyList<Double>。这将创建一个新类型，TMyList<Double>，其定义与 TMyList 相同，<T>只是定义中“T”的所有实例都替换为“Double”。

应该注意的是，泛型作为一种抽象机制复制了多态性的大部分功能，但具有不同的特征。由于新类型或方法是在实例化时构造的，因此可以在编译时而不是运行时更快地发现类型错误。这也增加了优化的范围，但需要权衡 - 每个实例化都会增加最终运行的应用程序的内存使用量，从而导致性能降低。

代码示例

例如，TSIPair 是一个包含两种数据类型（字符串和整数）的类：

```
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;
```

若要使类独立于数据类型，请将数据类型替换为类型参数：

```
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>;
```

平台要求和差异

泛型由 Delphi 编译器支持。

运行时类型标识

在 Win32 中，泛型和方法没有运行时类型信息 （RTTI），但实例化类型具有 RTTI。实例化类型是泛型与一组参数的组合。类方法的 RTTI 是整个类的 RTTI 的子集。如果非泛型类具有泛型方法，则该方法将不会包含在为该类生成的 RTTI 中，因为泛型是在编译时实例化的，而不是在运行时实例化的。

接口 GUID

在 Win32 中，实例化的接口类型没有接口 GUID。

接口中的参数化方法

参数化方法（使用类型参数声明的方法）不能在接口中声明。

实例化时序

泛型类型在编译时实例化，并发出到可执行文件和可重定位文件中。泛型类型的实例变量在运行时为类实例化，在编译时为泛型记录实例化。泛型类的 RTTI 仅在实例化类时生成。实例化类的 RTTI 与非泛型类一样。如果泛型类具有泛型方法，则实例化的泛型类将没有该泛型方法的 RTTI。

动态实例化

不支持运行时的动态实例化。

接口约束

Win32 接口不是“轻型”接口。这意味着具有接口约束的所有类型参数始终支持 COM IUnknown 方法_AddRef、_Release 和 QueryInterface 或从 TInterfacedObject 继承。记录类型不能指定接口约束参数。

术语

本节定义了用于描述泛型的术语

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.

声明泛型

泛型的声明类似于常规类、记录或接口类型的声明。区别在于，放置在尖括号（< 和 >）之间的一个或多个类型参数的列表跟在泛型声明中的类型标识符之后。

类型参数可用作容器类型声明和方法主体中的典型类型标识符。

例如：

```
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;
```

注：在调用 GetValue 方法之前，必须调用默认构造函数并初始化类字段。

类型参数

泛型类型通过提供类型参数进行实例化。在 Delphi 中，您可以使用任何类型作为类型参数，但以下类型除外：静态数组、短字符串或（递归）包含这两种类型中一种或多种字段的记录类型。

```
type
  TFoo<T> = class
    FData: T;
  end;
var
  F: TFoo<Integer>; // 'Integer' is the type argument of TFoo<T>
begin
  ...
end.
```

嵌套类型

泛型中的嵌套类型本身就是泛型。

```
type
  TFoo<T> = class                    // A generic class
  type
    TBar = class
      X: Integer;
    end;
  end;
```

若要访问 TBar 嵌套类型，必须先指定 TFoo 类型的构造：

```
 var
    N: TFoo<Double>.TBar;
```

泛型也可以在常规类中声明为嵌套类型：

```
Type
   TBaz = class                              // A regular class
   type
      TQux<T> = class
         X: Integer;
      end;
   end;
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;
```

基本类型

参数化类或接口类型的基类型可以是实际类型或构造类型。基类型可能不是单独的类型参数。

```
   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;
```

如果实例化了 TFoo2，则<String>祖先类将变为 TBar2<String>，并且 TBar2<String> 将自动实例化。

类、接口和记录类型

类、接口、记录和数组类型可以使用类型参数进行声明。
例如：

```
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>;
```

过程类型

可以使用类型参数声明过程类型和方法指针。参数类型和结果类型也可以使用类型参数。
例如：

```
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.
```

参数化方法

可以使用类型参数声明方法。参数类型和结果类型可以使用类型参数。但是，构造函数和析构函数不能有类型参数，虚拟、动态或消息方法也不能。参数化方法类似于重载方法。

有两种方法可以实例化方法：

- ✓ 显式指定类型参数

✓ 从类型参数自动推断

例如：

```
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.
```

类型参数的范围

类型参数的作用域涵盖类型声明及其所有成员的主体，但不包括后代类型。
例如：

```
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.
```

重载和类型兼容

重载

泛型方法可以使用“重载”指令与非泛型方法一起参与重载。如果泛型方法和非泛型方法之间的重载选择不明确，则编译器将选择非泛型方法。

例如：

```
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;
```

类型兼容

仅当两个非实例化泛型相同或与通用类型别名时，它们才被视为与赋值兼容。

如果基类型相同（或与通用类型别名）且类型参数相同，则两个实例化泛型被视为赋值兼容。

注意：Delphi 中的泛型与 C++ 中的模板或 C# 中的泛型类型不同。最值得注意的是，类型参数不能约束为特定的简单类型，如整数、双精度、字符串等。如果需要表示不同的类型（包括简单类型），请考虑对每个需要的类型使用重载函数，或者使用 System.Rtti 单元中的记录类型（如 TValue），该单元提供了用于存储和查询这些数据类型的运算符和方法。

泛型约束

约束可以与泛型的类型参数相关联。约束声明在泛型类型的构造中传递给该参数的任何特定类型必须支持的项。

使用约束指定泛型

约束项包括：

- ✓ 零、一种或多种接口类型
- ✓ 零个或一个类类型
- ✓ 保留字“构造函数”、“类”或“记录”

可以为约束指定“构造函数”和“类”。但是，“记录”不能与其他保留字组合使用。多个约束充当加法联合（“AND”逻辑）。

此处给出的示例仅显示类类型，尽管约束适用于所有形式的泛型。

声明约束

约束的声明方式类似于常规参数列表中的类型声明：

```
type
  TFoo<T: ISerializable> = class
    FField: T;
  end;
```

在此处给出的示例声明中，“T”类型参数指示它必须支持 ISerializable 接口。在像TFoo<TMyClass>这样的类型构造中，编译器在编译时进行检查，以确保 TMyClass 实际实现了ISerializable。

多种类型参数

指定约束时，可以用分号分隔多个类型参数，就像使用参数列表声明一样：

```
type
    TFoo<T: ISerializable; V: IComparable>
```

与参数声明一样，多个类型参数可以在逗号列表中组合在一起以绑定到相同的约束：

```
type
    TFoo<S, U: ISerializable> ...
```

在上面的示例中，S 和 U 都绑定到 ISerializedable 约束。

多重约束

可以将多个约束作为冒号后面的逗号列表应用于单个类型参数：

```
type
    TFoo<T: ISerializable, ICloneable; V: IComparable> ...
```

受约束的类型参数可以与“自由”类型参数混合使用。例如，以下所有内容均有效：

```
type
    TFoo<T; C: IComparable> ...
    TBar<T, V> ...
    TTest<S: ISerializable; V> ...
    // T and V are free, but C and S are constrained
```

约束类型

接口类型约束

类型参数约束可以包含多个接口类型的零个、一个或逗号分隔的列表。
受接口类型约束的类型参数意味着编译器将在编译时验证作为参数传递给类型构造的具体类型是否实现了指定的接口类型。

例如：

```
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
```

类类型约束

类型参数可以受零个或一个类类型的约束。与接口类型约束一样，此声明意味着编译器将要求作为参数传递给约束类型参数的任何具体类型都与约束类的赋值兼容。

类类型的兼容性遵循 OOP 类型兼容性的正常规则 - 后代类型可以在需要其祖先类型的地方传递。

构造函数约束

类型参数可以由保留字“constructor”的零个或一个实例约束。这意味着实际参数类型必须是定义默认构造函数（公共无参数构造函数）的类，以便泛型类型中的方法可以使用参数类型的默认构造函数构造参数类型的实例，而无需了解参数类型本身的任何信息（无最低基类型要求）。

在约束声明中，可以按任意顺序将“构造函数”与接口或类类型约束混合使用。

类约束

类型参数可以由保留字“class”的零个或一个实例约束。这意味着实际类型必须是类类型。

记录约束

类型参数可以受保留字“record”的零个或一个实例的约束。这意味着实际类型必须是值类型（而不是引用类型）。“记录”约束不能与“类”或“构造函数”约束结合使用。

类型推理

使用受约束类型参数的字段或变量时，在许多情况下无需类型转换即可将字段或变量视为受约束类型之一。编译器可以通过查看方法名称并通过在该类型的所有约束中共享相同名称的方法的联合执行重载解析的变化来推断所引用的类型。

例如：

```
type
  TFoo<T: ISerializable, ICloneable> = class
    FData: T;
    procedure Test;
  end;
```

```
procedure TFoo<T>.Test;
begin
  FData.Clone;
end;
```

编译器在 ISerializable 和 ICloneable 中查找“克隆”方法，因为 FData 属于 T 类型，可以保证支持这两个接口。如果两个接口都使用相同的参数列表实现“Clone”，编译器将发出不明确的方法调用错误，并要求您类型转换为一个或另一个接口以消除上下文的歧义。

类变量

泛型类型中定义的类变量在由类型参数标识的每个实例化类型中实例化。

以下代码显示 TFoo<Integer>.FCount and TFoo<String>.FCount 只实例化一次，这是两个不同的变量：

```
{$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.
```

14. 反射

提供属性概述：如何创建自定义属性、批注类型和成员、在运行时提取属性数据以及截获类的虚拟方法调用。

属性和反射

介绍属性的概念、属性的一般用例以及 Delphi 语言中的一些基本限制。

注意：C++生成器不支持 Delphi 属性。有关 C++Builder 中的 RTTI 的信息，请参阅 Delphi RTTI 和 C++Builder。

介绍

属性是 Delphi 中的一项语言功能，它允许使用携带附加信息的特殊对象来批注类型和类型成员。可以在运行时查询此信息。属性使用面向方面的元素扩展了普通的面向对象模型。

通常，在构建通用框架时，属性非常有用，这些框架在运行时分析结构化类型（如对象或记录），并根据批注属性提供的其他信息引入新行为。

属性和 RTTI

属性本身不会修改类型或成员的行为。使用者代码必须专门查询它们是否存在，并在需要时采取适当的操作。为了能够将属性附加到已编译二进制文件中的实体，首先需要为该实体发出 RTTI 信息。这意味着显式禁用 RTTI 信息的类型不符合属性注释的条件。例如，在下面的代码中，SomeCustomAttribute 不会发出到编译的二进制文件，因为 RTTI 信息专门禁用了 TDerivedObject 类。

```
type
  {$RTTI EXPLICIT METHODS([]) PROPERTIES([]) FIELDS([])}
  TDerivedObject = class(TObject)
    [SomeCustomAttribute()]
    procedure Shoot;
  end;
```

声明自定义属性

本主题介绍用于创建自定义属性的基本方法、属性类的正确设计决策以及常规用例。

声明属性

属性是一个简单的类类型。若要声明自己的自定义属性，必须从特殊的预定义类派生它：System.TCustomAttribute：

```
type
    MyCustomAttribute = class(TCustomAttribute)
    end;
```

然后，MyCustomAttribute 可用于批注任何类型或类型（如类、记录或接口）的任何成员：

```
type
  [MyCustomAttribute]
  TSpecialInteger = type integer;

  TSomeClass = class
      [MyCustomAttribute]
      procedure Work;
  end;
```

请注意，声明的属性类不得声明为类抽象，并且不应包含任何抽象方法。即使编译器允许您使用这些属性进行注释，构建的二进制文件也不会将它们包含在发出的 RTTI 信息中。

以“Attribute”结尾的属性名称被隐式缩短

假设您定义了两个具有相同名称前缀的 TCustomAttribute 子类，但其中一个子类具有“Attribute”作为后缀，例如：

```
MyCustom
MyCustomAttribute
```

始终使用带有“Attribute”后缀（MyCustomAttribute）的类，并且具有较短名称（MyCustom）的类变得不可访问。

以下代码片段演示了此问题。人们可能期望应用 TCustomAttribute 子类 Test，但由于隐式名称缩短，TestAttribute 实际上将应用于使用[Test]或[TestAttribute]的地方。

```
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;
```

属性中的构造函数

通常，属性旨在携带一些可在运行时查询的其他信息。若要允许为属性类指定自定义信息，必须为其声明构造函数：

```
type
    AttributeWithConstructor = class(TCustomAttribute)
    public
        constructor Create(const ASomeText: String);
    end;
```

然后可以按如下方式使用：

```
type
  [AttributeWithConstructor('Added text value!')]
  TRecord = record
        FField: Integer;
  end;
```

方法解析也适用于属性，这意味着您可以在自定义属性中定义重载构造函数。仅声明接受常量值的构造函数，而不声明 out 或 var 值。这源于属性工作方式的基本限制，在批注类型和类型成员中进行了更详细的讨论。

注解类型和类型成员

本主题介绍使用属性批注类型或成员时适用的语法和规则。

一般语法

若要批注 Delphi 类型或成员（如类或类成员），必须在该类型的声明前面加上括号之间的属性类的名称：

```
[CustomAttribute]
TMyClass = class;
```

如果属性类的名称以“属性”结尾，也可以省略“属性”后缀：

```
[Custom]
procedure DoSomething;
```

在属性类名后有一组括号也是一种有效的语法：

```
[Custom()]
TMyRecord = record;
```

某些属性接受参数。若要将参数传递给属性，请使用与方法调用相同的语法：

```
[Custom(Argument1, Argument2, …)]
TSimpleType = set of (stOne, stTwo, stThree);
```

若要批注具有多个属性的单个类型，可以使用多组括号：

```
[Custom1]
[Custom2(MyArgument)]
FString: String;
```

或者在一组括号之间使用逗号分隔的属性：

```
[Custom1, Custom2(MyArgument)]
function IsReady: Boolean;
```

只能将常量表达式用作属性参数

批注到类型或成员的属性将插入到生成的二进制文件中的 RTTI 信息块中。发出的信息包括：

- ✓ 属性的类类型。
- ✓ 指向所选构造函数的指针。
- ✓ 稍后传递给属性构造函数的常量列表。

传递给属性构造函数的值必须是常量表达式。由于这些值必须直接嵌入到生成的二进制文件中，因此无法传递需要运行时计算的表达式。这会对编译时可以传递给属性的信息产生一些限制：

- ✓ 只能使用常量表达式，包括集合、字符串和序号表达式。

- ✓ 您可以使用 TypeInfo（） 传递类型信息，因为 RTTI 块地址在编译时是已知的。
- ✓ 可以使用类引用，因为在编译时已知元类地址。
- ✓ 不能使用 out 或 var 参数，因为它们需要对传递参数的地址进行运行时计算。
- ✓ 你不能使用 Addr（） 或 @。

下面的代码举例说明了编译器不编译批注的情况：

```
var
    a, b: Integer;
type
    [SomeAttribute(a + b)]
    TSomeType = record
    // …
    end;
```

在前面的示例中，SomeAttribute 的构造函数需要一个整数值。传递的表达式需要 a + b 的运行时计算。编译器发出编译时错误，因为它需要一个常量表达式。

下面的代码显示了一个接受的表达式：

```
const
    a = 10;
    b = 20;
type
    [SomeAttribute(a + b)]
    TSomeType = record
    // …
    end;
```

a 和 b 的值在编译时是已知的;因此，直接计算常量表达式。

反射读取属性

提供有关属性的运行时方面的信息 - 如何提取属性以及如何根据其信息值做出自定义决策。

属性实例化

批注（如批注类型和类型成员中所述）是将属性附加到类型或成员的简单方法。编译二进制文件中包含的信息仅包括属性的类、指向所选构造函数的指针以及在实例化时传递给属性构造函数的常量列表。

当使用者代码在给定类型或类型成员中查询属性时，将发生属性的实际实例化。这意味着属性类的实例不是自动创建的，而是在程序显式搜索它们时创建的。没有保证属性实例化的顺

序，也不知道创建了多少个实例。程序不应该依赖于这样的后果。

请考虑以下属性声明：

```
type
  TSpecialAttribute = class(TCustomAttribute)
  public
    FValue: String;

    constructor Create(const AValue: String);
  end;

constructor TSpecialAttribute.Create(const AValue: String);
begin
  FValue := AValue;
end;
```

然后，在以下示例中将 TSpecialAttribute 用作注释：

```
type
  [TSpecialAttribute('Hello World!')]
  TSomeType = record
  ...
  end;
```

若要从 TSomeType 类型中提取属性，用户代码必须使用 System.Rtti 单元公开的功能。以下示例演示了提取代码：

```
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;
```

如上面的示例所示，用户必须专门编写代码来查询批注到类型的属性。实际的属性实例是在 TRttiType.GetAttributes 方法中创建的。请注意，该示例不会销毁实例;TRttiContext 之后会释放所有资源。

异常

由于属性的实际实例化是在用户代码中执行的，因此必须注意属性构造函数中可能发生的异常。一般建议使用 try..except 查询属性的代码周围的子句。

为了举例说明这个问题，原始示例中的属性构造函数更改为如下所示：

```
constructor TSpecialAttribute.Create(const AValue: String);
begin
  if AValue = '' then
    raise EArgumentException.Create('Expected a non-null string');

  FValue := AValue;
end;
```

并且 TSomeType 的注释更改为将空字符串传递给属性构造函数：

```
type
  [TSpecialAttribute('')]
  TSomeType = record
  ...
  end;
```

在这种情况下，查询 TSomeType 类型的属性的代码将失败，并出现由实例化属性引发的 EArgumentException 异常。建议更改查询代码以使用 try ..except：

```
  { 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;
```

使用虚拟方法拦截器

Delphi 在 Rtti.pas 中有一个名为 System.Rtti.TVirtualMethodInterceptor 的新类型。实质上，此类型在运行时动态创建一个派生元类，通过创建新的虚拟方法表并使用截获调用和参数的存根填充它，该元类覆盖祖先中的每个虚拟方法。当“祖先”的任何实例的元类引用被这个新的元类替换时，用户可以截获虚函数调用、动态更改参数、更改返回值、拦截和抑制异常或引发新异常，或者完全替换调用底层方法。

在概念上，这有点类似于来自.NET 和 Java 的动态代理。这就像能够在运行时从类派生，重写方法（但不添加新的实例字段），然后将实例的运行时类型更改为这个新的派生类。

有关更多信息，请参阅 Barry Kelly 的博客 http://blog.barrkel.com/2010/09/virtual-method-interception.html

编译器属性

一些特殊属性会触发 Delphi 编译器的某些功能。

Ref

Ref 属性用于限定常量函数参数，以便通过引用（而不是按值）将它们传递给函数。有关详细信息，请参阅常量参数。

Volatile

volatile 属性用于标记可能被不同线程更改的字段，以便代码生成不会优化在寄存器或其他临时内存位置复制值。

可以使用 volatile 属性来标记以下声明：

- ✓ 变量（全局和局部）
- ✓ 参数
- ✓ 记录或类的字段。

不能使用 volatile 属性来标记以下声明：

- ✓ 类型
- ✓ 过程、函数或方法
- ✓ 表达式

```
type
    TMyClass = class
    private
```

```
        [volatile] FMyVariable: TMyType;
    end;
```

Weak and Unsafe 属性

注意：弱引用和不安全引用的使用最初是作为移动平台的 ARC 内存管理支持的一部分引入的。由于 ARC 现已逐步淘汰，因此此功能仍仅适用于接口引用。

Weak

weak 是接口引用，它不会增加它们引用的对象的引用计数。它用 [weak] 修饰符标记。此功能仅由 Delphi 支持。在以下情况下，可以使用弱引用：

- ✓ 避免增加计数引用。
- ✓ 管理接口引用。这意味着当删除实际对象时，系统会跟踪它们。

注： 由于弱参照是完全管理的，因此如果被参照对象被销毁，它们将自动设置为 nil。

- ✓ 使用交叉引用断开两个对象之间的循环引用。发生这种情况时，您最终会出现内存泄漏。

例如，考虑以下接口接受对相同类型的另一个接口的引用，以及使用内部引用实现该接口的类：

```
type
  ISimpleInterface = interface
  procedure DoSomething;
  procedure AddObjectRef(Simple: ISimpleInterface);
 end;
 TObjectOne = class(TInterfacedObject, ISimpleInterface)
 private
  AnotherObj: ISimpleInterface;
 public
  procedure DoSomething;
  procedure AddObjectRef(Simple: ISimpleInterface);
 end;
```

如果创建两个对象并交叉引用它们，则最终会出现内存泄漏：

```
var
 One, Two: ISimpleInterface; begin
  One := TObjectOne.Create;
  Two := TObjectOne.Create;
  One.AddObjectRef(Two);
  Two.AddObjectRef(One);
```

现在 Delphi 中可用的解决方案是将私有字段 AnotherObj 标记为弱接口引用：

```
private
  [weak] AnotherObj: ISimpleInterface;
```

通过此更改，当您将对象作为参数传递给 AddObjectRef 调用时，引用计数不会修改，它保持为 1，当变量超出范围时，它返回到零，从而从内存中释放对象。
注意：您只能将 [弱] 变量传递给标记为 [弱] 的 var 或 out 参数。不能将常规强引用传递给 [弱] var 或 out 参数。

Unsafe

不安全引用是不增加它们所引用对象的引用计数的接口引用。它标有 [不安全] 修饰符。

注意：不安全引用不受管理，类似于没有额外编译器支持的指针。
如果要创建不在引用总数之外的接口引用，则可以使用 unsafe。例如：

```
procedure TForm3.Button2Click(Sender: TObject);
var
  [unsafe] OneIntf: ISimpleInterface;
begin
  OneIntf := TObjectOne.Create;
  OneIntf.DoSomething;
end;
```

注意：您只能将 [不安全] 变量传递给同样标记为 [不安全] 的 var 或 out 参数。不能将常规强引用传递给 [不安全] var 或 out。

附录一.Delphi 编译器指令

Delphi Compiler Directive	Associated Symbol
Align fields (Delphi)	{$A},{$ALIGN}
ALLOWBIND	{$ALLOWBIND}
ALLOWISOLATION	{$ALLOWISOLATION}
Application type (Delphi)	{$APPTYPE}
Assert directives (Delphi)	{$C},{$ASSERTIONS}
Boolean short-circuit evaluation (Delphi compiler directive)	{$B},{$BOOLEVAL}
Code align (Delphi)	{$CODEALIGN}

Compiler directives for libraries or shared objects (Delphi)	{$LIBPREFIX}, {$LIBSUFFIX}, {$LIBVERSION}
Conditional compilation (Delphi)	{$IFDEF}, {$IFNDEF}, {$IF}, {$ELSEIF}, {$ELSE}, {$ENDIF}, {$IFEND}
Data Execution Prevention compatible	{$NXCOMPAT}
Debug information (Delphi)	{$D},{$DEBUGINFO}
DEFINE directive (Delphi)	{$DEFINE}
DENYPACKAGEUNIT directive (Delphi)	{$DENYPACKAGEUNIT}
Description (Delphi)	{$D},{$DESCRIPTION}
DESIGNONLY directive (Delphi)	{$DESIGNONLY}
Dynamic base (Delphi)	{$DYNAMICBASE}
ELSE (Delphi)	{$ELSE}
ELSEIF (Delphi)	{$ELSEIF}
Enable Large Addresses	{$LARGEADDRESSAWARE}
ENDIF directive (Delphi)	{$ENDIF}
Executable extension (Delphi)	{$E},{$EXTENSION}
Export symbols (Delphi)	{$ObjExportAll}
Extended syntax (Delphi)	{$X},{$EXTENDEDSYNTAX}
Extended type compatibility (Delphi)	{$EXTENDEDCOMPATIBILITY}
External Symbols (Delphi)	{$EXTERNALSYM ['typeNameInHpp' ['typeNameInHppUnion']]}
Floating point precision control (Delphi for x64)	{$EXCESSPRECISION}
HIGHCHARUNICODE directive (Delphi)	{$HIGHCHARUNICODE}
Hints (Delphi)	{$HINTS}
HPP emit (Delphi)	{$HPPEMIT}
IFDEF directive (Delphi)	{$IFDEF}
IF directive (Delphi)	{$IF}
IFEND directive (Delphi)	{$IFEND}
IFNDEF directive (Delphi)	{$IFNDEF}
IFOPT directive (Delphi)	{$IFOPT}
Image base address	{$IMAGEBASE}
Implicit Build (Delphi)	{$IMPLICITBUILD}
Imported data	{$G},{$IMPORTEDDATA}

Include file (Delphi)	{$I},{$INCLUDE}
Input output checking (Delphi)	{$I},{$IOCHECKS}
Legacy IFEND (Delphi)	{$LEGACYIFEND}
Link object file (Delphi)	{$L file},{$LINK file}
Local symbol information (Delphi)	{$L+},{$LOCALSYMBOLS}
Long strings (Delphi)	{$H},{$LONGSTRINGS}
Memory allocation sizes (Delphi)	{$M},{$MINSTACKSIZE},{$MAXSTACKSIZE}
MESSAGE directive (Delphi)	{$MESSAGE }
METHODINFO directive (Delphi)	{$METHODINFO}
Minimum enumeration size (Delphi)	{$Z1},{$Z2},{$Z4},{$MINENUMSIZE 1},{$MINENUMSIZE 2},{$MINENUMSIZE 4}
NODEFINE directive (Delphi)	{$NODEFINE ['typeNameInHpp' ['typeNameInHppUnion']]}
NOINCLUDE (Delphi)	{$NOINCLUDE}
OBJTYPENAME directive (Delphi)	{$OBJTYPENAME typeIdent ['{B|N}typeNameInObj']}
Old type layout (Delphi)	{$OLDTYPELAYOUT ON}
Open String Parameters (Delphi)	{$P},{$OPENSTRINGS}
Optimization (Delphi)	{$O},{$OPTIMIZATION}
Overflow checking (Delphi)	{$Q},{$OVERFLOWCHECKS}
PE (portable executable) header flags (Delphi)	{$SetPEFlags},{$SetPEOptFlags}
PE header operating system version	{$SETPEOSVERSION}
PE header subsystem version	{$SETPESUBSYSVERSION}
PE header user version	{$SETPEUSERVERSION}
Pentium-safe FDIV operations (Delphi)	{$U},{$SAFEDIVIDE}
Pointer Math (Delphi)	{$POINTERMATH}
Range checking	{$R},{$RANGECHECKS}
Real48 compatibility (Delphi)	{$REALCOMPATIBILITY}
Regions	{$REGION},{$ENDREGION}
Reserved address space for resources (Delphi, Linux)	{$M},{$RESOURCERESERVE}
Resource file (Delphi)	{$R},{$RESOURCE}
RTTI directive (Delphi)	{$RTTI INHERIT|EXPLICIT}
RUNONLY directive (Delphi)	{$RUNONLY}

Run-Time Type Information (Delphi)	{$M},{$TYPEINFO}
Scoped Enums (Delphi)	{$SCOPEDENUMS}
Stack frames (Delphi)	{$W},{$STACKFRAMES}
Strong link types (Delphi)	{$STRONGLINKTYPES}
Support high-entropy 64-bit ASLR	{$HIGHENTROPYVA}
Symbol declaration and cross-reference information (Delphi)	{$Y},{$REFERENCEINFO},{DEFINITIONINFO}
Terminal Server Aware	{$TSAWARE}
Type-checked pointers (Delphi)	{$T},{$TYPEDADDRESS}
UNDEF directive (Delphi)	{$UNDEF}
Var-string checking (Delphi)	{$V},{$VARSTRINGCHECKS}
Warning messages (Delphi)	{$WARN}
Warnings (Delphi)	{$WARNINGS}
Weak packaging	{$WEAKPACKAGEUNIT}
WEAKLINKRTTI directive (Delphi)	{$WEAKLINKRTTI}
Writeable typed constants (Delphi)	{$J},{$WRITEABLECONST}
Zero-based strings (Delphi)	{$ZEROBASEDSTRINGS}

附录二.Delphi 编译器警告和错误

运行时错误

运行时的某些错误会导致 Delphi 程序显示错误消息并终止。

识别运行时错误

运行时错误的形式如下：

Run-time error nnn at xxxxxxxx

其中 nnn 是运行时错误号，xxxxxxxx 是运行时错误地址。
使用 SysUtils 类的应用程序将大多数运行时错误映射到异常，这使应用程序可以在不终止的情况下解决错误。

运行时错误的类型

Delphi 运行时错误分为以下几类：

- ✓ I/O 错误，编号为 100 到 149
- ✓ 程序退出错误，编号为 200 到 255
- ✓ 操作系统错误

I/O 错误

I/O（输入-输出）错误会导致在语句以 {$I+} 状态编译时引发异常。（如果应用程序不包含 System.SysUtils 单元，则异常会导致应用程序终止）。

处理 I/O 错误

在 {$I-} 状态下，程序继续执行，并且 IOResult 函数报告错误。

I/O 错误列表

下表列出了所有 I/O 错误、编号和说明。

Number	Name	Description
100	Disk read error	Reported by Read on a typed file if you attempt to read past the end of the file.
101	Disk write error	Reported by CloseFile, Write, WriteIn, or Flush if the disk becomes full.
102	File not assigned	Reported by Reset, Rewrite, Append, Rename, or Erase if the file variable has not been assigned a name through a call to Assign or AssignFile.
103	File not open	Reported by CloseFile, Read Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead, or BlockWrite if the file is not open.
104	File not open for input	Reported by Read, Readln, Eof, Eoln, SeekEof, or SeekEoln on a text file if the file is not open for input.
105	File not open for output	Reported by Write or Writeln on a text file if you do not generate a Console application.
106	Invalid numeric format	Reported by Read or Readln if a numeric value read from a text file does not conform to the proper numeric format.

操作系统错误

除 I/O 错误和程序退出错误之外的所有错误都使用操作系统返回的错误代码进行报告。

操作系统错误代码

操作系统错误代码是操作系统函数调用的返回值。您可以通过调用全局 System.GetLastError 函数获取发生的最后一个错误的错误代码。如果要引发异常，而不是获取上次失败的 API 调用的错误代码，请改为调用 SysUtils.RaiseLastOSError 过程。

GetLastError 返回的错误代码值取决于操作系统。可以通过调用全局 System.SysUtils.SysErrorMessage 函数来获取与这些错误代码之一关联的错误字符串。

获取返回值

若要检查来自 Win32 API 函数调用的返回值，并在表示错误时引发 EWin32Error 异常，请调用全局 System.SysUtils.Win32Check 函数。

程序终止错误

致命错误总是立即终止程序。

异常映射

在使用 System.SysUtils 单元的应用程序中（与大多数 GUI 应用程序一样），致命错误将映射到异常。有关生成每个错误的条件的说明，请参阅异常的文档。

I/O 错误列表

下表列出了所有致命错误、数字和映射的异常。

Number	Name	Exception
200	Division by zero	System.SysUtils.EDivByZero
201	Range check error	System.ERangeError
202	Stack overflow	EStackOverflow
203	Heap overflow error	EOutOfMemory
204	Invalid pointer operation	EInvalidPointer
205	Floating point overflow	System.EOverflow
206	Floating point underflow	System.EUnderflow
207	Invalid floating point operation	System.EInvalidOp

210	Abstract Method Error	EAbstractError
215	Arithmetic overflow (integer only)	System.EIntOverflow
216	Access violation	EAccessViolation
217	Control-C	EControlC
218	Privileged instruction	EPrivilege
219	Invalid typecast	System.EInvalidCast
220	Invalid variant typecast	Variants.EVariantError
221	Invalid variant operation	Variants.EVariantError
222	No variant method call dispatcher	Variants.EVariantError
223	Cannot create variant array	Variants.EVariantError
224	Variant does not contain array	Variants.EVariantError
225	Variant array bounds error	Variants.EVariantError
226	TLS initialization error	No exception to map to.
227	Assertion failed	System.SysUtils.EAssertionFailed
228	Interface Cast Error	System.SysUtils.EIntfCastError
229	Safecall error	System. SysUtils.ESafeCallException
230	Unhandled exception	No exception to map to.
231	Too many nested exceptions	Up to 16 permitted.
232	Fatal signal raised on a non-Delphi thread	No exception to map to.

编译器警告和错误

错误消息

DisposeCount cannot be declared in classes with destructors (Delphi)
E1038 Unit identifier '%s' does not match file name (Delphi)
E2001 Ordinal type required (Delphi)
E2002 File type not allowed here (Delphi)
E2003 Undeclared identifier '%s' (Delphi)
E2004 Identifier redeclared '%s' (Delphi)
E2005 '%s' is not a type identifier (Delphi)
E2006 PACKED not allowed here (Delphi)
E2007 Constant or type identifier expected (Delphi)
E2008 Incompatible types (Delphi)
E2009 Incompatible types - '%s' (Delphi)
E2010 Incompatible types - '%s' and '%s' (Delphi)
E2011 Low bound exceeds high bound (Delphi)
E2012 Type of expression must be BOOLEAN (Delphi)
E2013 Type of expression must be INTEGER (Delphi)

E2014 Statement expected, but expression of type '%s' found (Delphi)
E2015 Operator not applicable to this operand type (Delphi)
E2016 Array type required (Delphi)
E2017 Pointer type required (Delphi)
E2018 Record, object or class type required (Delphi)
E2019 Object type required (Delphi)
E2020 Object or class type required (Delphi)
E2021 Class type required (Delphi)
E2022 Class helper type required (Delphi)
E2023 Function needs result type (Delphi)
E2024 Invalid function result type (Delphi)
E2025 Procedure cannot have a result type (Delphi)
E2026 Constant expression expected (Delphi)
E2027 Duplicate tag value (Delphi)
E2028 Sets may have at most 256 elements (Delphi)
E2029 %s expected but %s found (Delphi)
E2030 Duplicate case label (Delphi)
E2031 Label expected (Delphi)
E2032 For loop control variable must have ordinal type (Delphi)
E2033 Types of actual and formal var parameters must be identical (Delphi)
E2034 Too many actual parameters (Delphi)
E2035 Not enough actual parameters (Delphi)
E2036 Variable required (Delphi)
E2037 Declaration of '%s' differs from previous declaration (Delphi)
E2038 Illegal character in input file - '%s' (%s) (Delphi)
E2045 Bad object file format - '%s' (Delphi)
E2049 Label declaration not allowed in interface part (Delphi)
E2050 Statements not allowed in interface part (Delphi)
E2052 Unterminated string (Delphi)
E2053 Syntax error in real number (Delphi)
E2054 Illegal type in Write/Writeln statement (Delphi)
E2055 Illegal type in Read/Readln statement (Delphi)
E2056 String literals may have at most 255 elements (Delphi)
E2057 Unexpected end of file in comment started on line %ld (Delphi)
E2058 Class, interface and object types only allowed in type section (Delphi)
E2059 Local class, interface or object types not allowed (Delphi)
E2060 Class and interface types only allowed in type section (Delphi)
E2061 Local class or interface types not allowed (Delphi)
E2062 Virtual constructors are not allowed (Delphi)
E2064 Left side cannot be assigned to (Delphi)
E2065 Unsatisfied forward or external declaration '%s' (Delphi)
E2066 Missing operator or semicolon (Delphi)
E2067 Missing parameter type (Delphi)
E2068 Illegal reference to symbol '%s' in object file '%s' (Delphi)

E2070 Unknown directive - '%s' (Delphi)
E2071 This type cannot be initialized (Delphi)
E2072 Number of elements (%d) differs from declaration (%d) (Delphi)
E2073 Label already defined '%s' (Delphi)
E2074 Label declared and referenced, but not set '%s' (Delphi)
E2075 This form of method call only allowed in methods of derived types (Delphi)
E2076 This form of method call only allowed for class methods (Delphi)
E2078 Procedure FAIL only allowed in constructor (Delphi)
E2079 Procedure NEW needs constructor (Delphi)
E2080 Procedure DISPOSE needs destructor (Delphi)
E2081 Assignment to FOR-Loop variable '%s' (Delphi)
E2082 TYPEOF can only be applied to object types with a VMT (Delphi)
E2083 Order of fields in record constant differs from declaration (Delphi)
E2085 Unit name mismatch: '%s' '%s' (Delphi)
E2086 Type '%s' is not yet completely defined (Delphi)
E2088 Variable name expected (Delphi)
E2089 Invalid typecast (Delphi)
E2090 User break - compilation aborted (Delphi)
E2091 Segment/Offset pairs not supported in 32-bit Delphi (Delphi)
E2093 Label '%s' is not declared in current procedure (Delphi)
E2094 Local procedure/function '%s' assigned to procedure variable (Delphi)
E2095 Missing ENDIF directive (Delphi)
E2096 Method identifier expected (Delphi)
E2097 BREAK or CONTINUE outside of loop (Delphi)
E2098 Division by zero (Delphi)
E2099 Overflow in conversion or arithmetic operation (Delphi)
E2100 Data type too large exceeds 2 GB (Delphi)
E2102 Integer constant too large (Delphi)
E2103 16-Bit fixup encountered in object file '%s' (Delphi)
E2104 Bad relocation encountered in object file '%s' (Delphi)
E2105 Inline assembler syntax error (Delphi)
E2106 Inline assembler stack overflow (Delphi)
E2107 Operand size mismatch (Delphi)
E2108 Memory reference expected (Delphi)
E2109 Constant expected (Delphi)
E2110 Type expected (Delphi)
E2111 Cannot add or subtract relocatable symbols (Delphi)
E2112 Invalid register combination (Delphi)
E2113 Numeric overflow (Delphi)
E2114 String constant too long (Delphi)
E2115 Error in numeric constant (Delphi)
E2116 Invalid combination of opcode and operands (Delphi)
E2117 486/487 instructions not enabled (Delphi)
E2118 Division by zero (Delphi)

E2119 Structure field identifier expected (Delphi)
E2120 LOOP/JCXZ distance out of range (Delphi)
E2121 Procedure or function name expected (Delphi)
E2122 PROCEDURE or FUNCTION expected (Delphi)
E2123 PROCEDURE, FUNCTION, PROPERTY, or VAR expected (Delphi)
E2124 Instance member '%s' inaccessible here (Delphi)
E2125 EXCEPT or FINALLY expected (Delphi)
E2126 Cannot BREAK, CONTINUE or EXIT out of a FINALLY clause (Delphi)
E2127 'GOTO %s' leads into or out of TRY statement (Delphi)
E2128 %s clause expected, but %s found (Delphi)
E2129 Cannot assign to a read-only property (Delphi)
E2130 Cannot read a write-only property (Delphi)
E2131 Class already has a default property (Delphi)
E2132 Default property must be an array property (Delphi)
E2133 TYPEINFO standard function expects a type identifier (Delphi)
E2134 Type '%s' has no type info (Delphi)
E2136 No definition for abstract method '%s' allowed (Delphi)
E2137 Method '%s' not found in base class (Delphi)
E2138 Invalid message parameter list (Delphi)
E2139 Illegal message method index (Delphi)
E2140 Duplicate message method index (Delphi)
E2142 Inaccessible value (Delphi)
E2143 Expression has no value (Delphi)
E2144 Destination is inaccessible (Delphi)
E2145 Re-raising an exception only allowed in exception handler (Delphi)
E2146 Default values must be of ordinal, pointer or small set type (Delphi)
E2147 Property '%s' does not exist in base class (Delphi)
E2148 Dynamic method or message handler not allowed here (Delphi)
E2149 Class does not have a default property (Delphi)
E2150 Bad argument type in variable type array constructor (Delphi)
E2151 Could not load RLINK32.DLL (Delphi)
E2152 Wrong or corrupted version of RLINK32.DLL (Delphi)
E2153 ';' not allowed before 'ELSE' (Delphi)
E2154 Type '%s' needs finalization - not allowed in variant record (Delphi)
E2155 Type '%s' needs finalization - not allowed in file type (Delphi)
E2156 Expression too complicated (Delphi)
E2157 Element 0 inaccessible - use 'Length' or 'SetLength' (Delphi)
E2158 %s unit out of date or corrupted - missing '%s' (Delphi)
E2159 %s unit out of date or corrupted - missing '%s.%s' (Delphi)
E2160 Type not allowed in OLE Automation call (Delphi)
E2163 Too many conditional symbols (Delphi)
E2165 Compile terminated by user (Delphi)
E2166 Unnamed arguments must precede named arguments in OLE Automation call (Delphi)
E2167 Abstract methods must be virtual or dynamic (Delphi)

E2168 Field or method identifier expected (Delphi)
E2169 Field definition not allowed after methods or properties (Delphi)
E2170 Cannot override a non-virtual method (Delphi)
E2171 Variable '%s' inaccessible here due to optimization (Delphi)
E2172 Necessary library helper function was eliminated by linker (%s) (Delphi)
E2173 Missing or invalid conditional symbol in '$%s' directive (Delphi)
E2174 '%s' not previously declared as a PROPERTY (Delphi)
E2175 Field definition not allowed in OLE automation section (Delphi)
E2176 Illegal type in OLE automation section: '%s' (Delphi)
E2177 Constructors and destructors not allowed in OLE automation section (Delphi)
E2178 Dynamic methods and message handlers not allowed in OLE automation section (Delphi)
E2179 Only register calling convention allowed in OLE automation section (Delphi)
E2180 Dispid '%d' already used by '%s' (Delphi)
E2181 Redeclaration of property not allowed in OLE automation section (Delphi)
E2182 '%s' clause not allowed in OLE automation section (Delphi)
E2183 Dispid clause only allowed in OLE automation section (Delphi)
E2184 %s section valid only in class types (Delphi)
E2185 Overriding automated virtual method '%s' cannot specify a dispid (Delphi)
E2186 Published Real property '%s' must be Single, Real, Double or Extended (Delphi)
E2187 Size of published set '%s' is 4 bytes (Delphi)
E2188 Published property '%s' cannot be of type %s (Delphi)
E2189 Thread local variables cannot be local to a function (Delphi)
E2190 Thread local variables cannot be ABSOLUTE (Delphi)
E2191 EXPORTS allowed only at global scope (Delphi)
E2192 Constants cannot be used as open array arguments (Delphi)
E2193 Slice standard function only allowed as open array argument (Delphi)
E2194 Cannot initialize thread local variables (Delphi)
E2195 Cannot initialize local variables (Delphi)
E2196 Cannot initialize multiple variables (Delphi)
E2197 Constant object cannot be passed as var parameter (Delphi)
E2198 %s cannot be applied to a long string (Delphi)
E2199 Packages '%s' and '%s' both contain unit '%s' (Delphi)
E2200 Package '%s' already contains unit '%s' (Delphi)
E2201 Need imported data reference ($G) to access '%s' from unit '%s' (Delphi)
E2202 Required package '%s' not found (Delphi)
E2203 $WEAKPACKAGEUNIT '%s' contains global data (Delphi)
E2204 Improper GUID syntax (Delphi)
E2205 Interface type required (Delphi)
E2206 Property overrides not allowed in interface type (Delphi)
E2207 '%s' clause not allowed in interface type (Delphi)
E2208 Interface '%s' already implemented by '%s' (Delphi)
E2209 Field declarations not allowed in interface type (Delphi)
E2210 '%s' directive not allowed in in interface type (Delphi)

E2211 Declaration of '%s' differs from declaration in interface '%s' (Delphi)
E2212 Package unit '%s' cannot appear in contains or uses clauses (Delphi)
E2213 Bad packaged unit format %s.%s (Delphi)
E2214 Package '%s' is recursively required (Delphi)
E2215 16-Bit segment encountered in object file '%s' (Delphi)
E2216 Can't handle section '%s' in object file '%s' (Delphi)
E2217 Published field '%s' not a class or interface type (Delphi)
E2218 Published method '%s' contains an unpublishable type (Delphi)
E2220 Never-build package '%s' requires always-build package '%s' (Delphi)
E2221 $WEAKPACKAGEUNIT '%s' cannot have initialization or finalization code (Delphi)
E2222 $WEAKPACKAGEUNIT & $DENYPACKAGEUNIT both specified (Delphi)
E2223 $DENYPACKAGEUNIT '%s' cannot be put into a package (Delphi)
E2224 $DESIGNONLY and $RUNONLY only allowed in package unit (Delphi)
E2225 Never-build package '%s' must be recompiled (Delphi)
E2226 Compilation terminated; too many errors (Delphi)
E2227 Imagebase is too high - program exceeds 2 GB limit (Delphi)
E2228 A dispinterface type cannot have an ancestor interface (Delphi)
E2229 A dispinterface type requires an interface identification (Delphi)
E2230 Methods of dispinterface types cannot specify directives (Delphi)
E2231 '%s' directive not allowed in dispinterface type (Delphi)
E2232 Interface '%s' has no interface identification (Delphi)
E2233 Property '%s' inaccessible here (Delphi)
E2234 Getter or setter for property '%s' cannot be found (Delphi)
E2236 Constructors and destructors must have %s calling convention (Delphi)
E2237 Parameter '%s' not allowed here due to default value (Delphi)
E2238 Default value required for '%s' (Delphi)
E2239 Default parameter '%s' must be by-value or const (Delphi)
E2240 $EXTERNALSYM and $NODEFINE not allowed for '%s'; only global symbols (Delphi)
E2241 C++ obj files must be generated (-jp) (Delphi)
E2242 '%s' is not the name of a unit (Delphi)
E2245 Recursive include file %s (Delphi)
E2246 Need to specify at least one dimension for SetLength of dynamic array (Delphi)
E2247 Cannot take the address when compiling to byte code (Delphi)
E2248 Cannot use old style object types when compiling to byte code (Delphi)
E2249 Cannot use absolute variables when compiling to byte code (Delphi)
E2250 There is no overloaded version of '%s' that can be called with these arguments (Delphi)
E2251 Ambiguous overloaded call to '%s' (Delphi)
E2252 Method '%s' with identical parameters already exists (Delphi)
E2253 Ancestor type '%s' does not have an accessible default constructor (Delphi)
E2254 Overloaded procedure '%s' must be marked with the 'overload' directive (Delphi)
E2255 New not supported for dynamic arrays - use SetLength (Delphi)
E2256 Dispose not supported (nor necessary) for dynamic arrays (Delphi)
E2257 Duplicate implements clause for interface '%s' (Delphi)
E2258 Implements clause only allowed within class types (Delphi)

E2259 Implements clause only allowed for properties of class or interface type (Delphi)
E2260 Implements clause not allowed together with index clause (Delphi)
E2261 Implements clause only allowed for readable property (Delphi)
E2262 Implements getter must be %s calling convention (Delphi)
E2263 Implements getter cannot be dynamic or message method (Delphi)
E2264 Cannot have method resolutions for interface '%s' (Delphi)
E2265 Interface '%s' not mentioned in interface list (Delphi)
E2266 Only one of a set of overloaded methods can be published (Delphi)
E2267 Previous declaration of '%s' was not marked with the 'overload' directive (Delphi)
E2268 Parameters of this type cannot have default values (Delphi)
E2270 Published property getters and setters must have %s calling convention (Delphi)
E2271 Property getters and setters cannot be overloaded (Delphi)
E2272 Cannot use reserved unit name '%s' (Delphi)
E2273 No overloaded version of '%s' with this parameter list exists (Delphi)
E2274 property attribute 'label' cannot be used in dispinterface (Delphi)
E2275 property attribute 'label' cannot be an empty string (Delphi)
E2276 Identifier '%s' cannot be exported (Delphi)
E2277 Only external cdecl functions may use varargs (Delphi)
E2278 Cannot take address of local symbol %s (Delphi)
E2279 Too many nested conditional directives (Delphi)
E2280 Unterminated conditional directive (Delphi)
E2281 Type not allowed in Variant Dispatch call (Delphi)
E2282 Property setters cannot take var parameters (Delphi)
E2283 Too many local constants. Use shorter procedures (Delphi)
E2284 Duplicate resource name type %d '%s' (Delphi)
E2285 Duplicate resource id type %d id %d (Delphi)
E2286 Coverage library name is too long: %s (Delphi)
E2287 Cannot export '%s' multiple times (Delphi)
E2288 File name too long (exceeds %d characters) (Delphi)
E2289 Unresolved custom attribute %s (Delphi)
E2290 Cannot mix destructors with IDisposable (Delphi)
E2291 Missing implementation of interface method %s.%s (Delphi)
E2292 '%s' must reference a property or field of class '%s' (Delphi)
E2293 Cannot have both a DLLImport attribute and an external or calling convention directive (Delphi)
E2294 A class helper that descends from '%s' can only help classes that are descendents '%s' (Delphi)
E2295 A class helper cannot introduce a destructor (Delphi)
E2296 A constructor introduced in a class helper must call the parameterless constructor of the helped class as the first statement (Delphi)
E2298 read/write not allowed for CLR events. Use Include/Exclude procedure (Delphi)
E2299 Property required (Delphi)
E2300 Cannot generate property accessor '%s' because '%s' already exists (Delphi)
E2301 Method '%s' with identical parameters and result type already exists (Delphi)

E2306 'Self' is initialized more than once (Delphi)

E2307 NEW standard function expects a dynamic array type identifier (Delphi)

E2308 Need to specify at least one dimension for NEW of dynamic array (Delphi)

E2309 Attribute - Known attribute named argument cannot be an array (Delphi)

E2310 Attribute - A custom marshaler requires the custom marshaler type (Delphi)

E2311 Attribute - MarshalAs fixed string requires a size (Delphi)

E2312 Attribute - Invalid argument to a known attribute (Delphi)

E2313 Attribute - Known attribute cannot specify properties (Delphi)

E2314 Attribute - The MarshalAs attribute has fields set that are not valid for the specified unmanaged type (Delphi)

E2315 Attribute - Known custom attribute on invalid target (Delphi)

E2316 Attribute - The format of the GUID was invalid (Delphi)

E2317 Attribute - Known custom attribute had invalid value (Delphi)

E2318 Attribute - The MarshalAs constant size cannot be negative (Delphi)

E2319 Attribute - The MarshalAs parameter index cannot be negative (Delphi)

E2320 Attribute - The specified unmanaged type is only valid on fields (Delphi)

E2321 Attribute - Known custom attribute has repeated named argument (Delphi)

E2322 Attribute - Unexpected type in known attribute (Delphi)

E2323 Attribute - Unrecognized argument to a known custom attribute (Delphi)

E2324 Attribute - Known attribute named argument doesn't support variant (Delphi)

E2325 Attribute '%s' is not valid on this target (Delphi)

E2326 Attribute '%s' can only be used once per target (Delphi)

E2327 Linker error while emitting attribute '%s' for '%s' (Delphi)

E2328 Linker error while emitting metadata (Delphi)

E2329 Metadata - Error occured during a read (Delphi)

E2330 Metadata - Error occured during a write (Delphi)

E2331 Metadata - File is read only (Delphi)

E2332 Metadata - An ill-formed name was given (Delphi)

E2333 Metadata - Data value was truncated (Delphi)

E2334 Metadata - Old version error (Delphi)

E2335 Metadata - A shared mem open failed to open at the originally (Delphi)

E2336 Metadata - Create of shared memory failed. A memory mapping of the same name already exists (Delphi)

E2337 Metadata - There isn't .CLB data in the memory or stream (Delphi)

E2338 Metadata - Database is read only (Delphi)

E2339 Metadata - The importing scope is not compatible with the emitting scope (Delphi)

E2340 Metadata - Data too large (Delphi)

E2341 Metadata - Column cannot be changed (Delphi)

E2342 Metadata - Too many RID or primary key columns, 1 is max (Delphi)

E2343 Metadata - Primary key column may not allow the null value (Delphi)

E2344 Metadata - Data too large (Delphi)

E2345 Metadata - Attempt to define an object that already exists (Delphi)

E2346 Metadata - A guid was not provided where one was required (Delphi)

E2347 Metadata - Bad binary signature (Delphi)

E2348 Metadata - Bad input parameters (Delphi)
E2349 Metadata - Cannot resolve typeref (Delphi)
E2350 Metadata - No logical space left to create more user strings (Delphi)
E2351 Final methods must be virtual or dynamic (Delphi)
E2352 Cannot override a final method (Delphi)
E2353 Cannot extend sealed class '%s' (Delphi)
E2354 String element cannot be passed to var parameter (Delphi)
E2355 Class property accessor must be a class field or class static method (Delphi)
E2356 Property accessor must be an instance field or method (Delphi)
E2357 PROCEDURE, FUNCTION, or CONSTRUCTOR expected (Delphi)
E2358 Class constructors not allowed in class helpers (Delphi)
E2359 Multiple class constructors in class %s: %s and %s (Delphi)
E2360 Class constructors cannot have parameters (Delphi)
E2361 Cannot access private symbol %s.%s (Delphi)
E2362 Cannot access protected symbol %s.%s (Delphi)
E2363 Only methods of descendent types may access protected symbol %s%s.%s across assembly boundaries (Delphi)
E2366 Global procedure or class static method expected (Delphi)
E2370 Cannot use inherited methods for interface property accessors (Delphi)
E2371 ABSTRACT and FINAL cannot be used together (Delphi)
E2372 Identifier expected (Delphi)
E2373 Call to abstract method %s.%s (Delphi)
E2374 Cannot make unique type from %s (Delphi)
E2375 PRIVATE or PROTECTED expected (Delphi)
E2376 STATIC can only be used on non-virtual class methods (Delphi)
E2377 Unable to locate Borland.Delphi.Compiler.ResCvt.dll (Delphi)
E2378 Error while converting resource %s (Delphi)
E2379 Virtual methods not allowed in record types (Delphi)
E2380 Instance or class static method expected (Delphi)
E2381 Resource string length exceeds Windows limit of 4096 characters (Delphi)
E2382 Cannot call constructors using instance variables (Delphi)
E2383 ABSTRACT and SEALED cannot be used together (Delphi)
E2385 Error while signing assembly (Delphi)
E2386 Invalid version string '%s' specified in %s (Delphi)
E2387 The key container name '%s' does not exist (Delphi)
E2388 Unrecognized strong name key file '%s' (Delphi)
E2389 Protected member '%s' is inaccessible here (Delphi)
E2390 Class must be sealed to call a private constructor without a type qualifier (Delphi)
E2391 Potentially polymorphic constructor calls must be virtual (Delphi)
E2392 Can't generate required accessor method(s) for property %s.%s due to name conflict with existing symbol %s in the same scope (Delphi)
E2393 Invalid operator declaration (Delphi)
E2394 Parameterless constructors not allowed on record types (Delphi)
E2395 Unsafe procedure only allowed if compiling with $UNSAFECODE ON (Delphi)

E2396 Unsafe code only allowed in unsafe procedure (Delphi)

E2397 Unsafe pointer only allowed if compiling with $UNSAFECODE ON (Delphi)

E2398 Class methods in record types must be static (Delphi)

E2399 Namespace conflicts with unit name '%s' (Delphi)

E2400 Unknown Resource Format '%s' (Delphi)

E2402 Constructing instance of abstract class '%s' (Delphi)

E2403 Add or remove accessor for event '%s' cannot be found (Delphi)

E2404 Cannot mix READ/WRITE property accessors with ADD/REMOVE accessors (Delphi)

E2405 Unknown element type found importing signature of %s.%s (Delphi)

E2406 EXPORTS section allowed only if compiling with $UNSAFECODE ON (Delphi)

E2407 Duplicate resource identifier %s found in unit %s(%s) and %s(%s) (Delphi)

E2408 Can't extract strong name key from assembly %s (Delphi)

E2409 Fully qualified nested type name %s exceeds 1024 byte limit (Delphi)

E2410 Unsafe pointer variables, parameters or consts only allowed in unsafe procedure (Delphi)

E2411 Unit %s in package %s refers to unit %s which is not found in any package. Packaged units must refer only to packaged units (Delphi)

E2412 CREATE expected (Delphi)

E2413 Dynamic array type needed (Delphi)

E2414 Disposed_ cannot be declared in classes with destructors (Delphi)

E2415 Could not import assembly '%s' because it contains namespace '%s' (Delphi)

E2416 Could not import package '%s' because it contains system unit '%s' (Delphi)

E2417 Field offset cannot be determined for variant record because previous field type is unknown size record type (Delphi)

E2418 Type '%s' needs initialization - not allowed in variant record (Delphi)

E2419 Record type too large: exceeds 1 MB (Delphi)

E2420 Interface '%s' used in '%s' is not yet completely defined (Delphi)

E2422 Imported identifier '%s' conflicts with '%s' in namespace '%s' (Delphi)

E2423 Void type not usable in this context (Delphi)

E2424 Codepage '%s' is not installed on this machine (Delphi)

E2425 Inline methods must not be virtual nor dynamic (Delphi)

E2426 Inline function must not have asm block (Delphi)

E2428 Field '%s' needs initialization - not allowed in CLS compliant value types (Delphi)

E2429 Duplicate implementation for 'set of %s' in this scope (Delphi)

E2430 for-in statement cannot operate on collection type '%s' (Delphi)

E2431 for-in statement cannot operate on collection type '%s' because '%s' does not contain a member for '%s', or it is inaccessible (Delphi)

E2432 %s cannot be applied to a rectangular dynamic array (Delphi)

E2433 Method declarations not allowed in anonymous record or local record type (Delphi)

E2434 Property declarations not allowed in anonymous record or local record type (Delphi)

E2435 Class member declarations not allowed in anonymous record or local record type (Delphi)

E2436 Type declarations not allowed in anonymous record or local record type (Delphi)

E2437 Constant declarations not allowed in anonymous record or local record type (Delphi)

E2439 Inline function must not have open array argument (Delphi)

E2441 Inline function declared in interface section must not use local symbol '%s' (Delphi)

E2442 Inline directive not allowed in constructor or destructor (Delphi)

E2447 Duplicate symbol '%s' defined in namespace '%s' by '%s' and '%s' (Delphi)

E2448 An attribute argument must be a constant expression, typeof expression or array constructor (Delphi)

E2449 Inlined nested routine '%s' cannot access outer scope variable '%s' (Delphi)

E2450 There is no overloaded version of array property '%s' that can be used with these arguments (Delphi)

E2452 Unicode characters not allowed in published symbols (Delphi)

E2453 Destination cannot be assigned to (Delphi)

E2454 Slice standard function not allowed for VAR nor OUT argument (Delphi)

E2459 Class property accessor must not have field selector (Delphi)

E2460 Cannot inherit from special class '%s.%s' (Delphi)

E2461 Identifier '%s' is not an instance field (Delphi)

E2463 '%s' directive not allowed in record type (Delphi)

E2464 First parameter of Assign operator must be var parameter (Delphi)

E2465 A record cannot introduce a destructor (Delphi)

E2466 Never-build package '%s' requires always-build package '%s' (Delphi)

E2467 Record or object type required (Delphi)

E2468 Linker detected duplicate name '%s' made from type '%s' in both unit '%s' and '%s' (Delphi)

E2469 Linker detected duplicate name '%s' made from both type '%s' in unit '%s' and type '%s' in unit '%s' (Delphi)

E2470 Linker detected duplicate name '%s' made from type '%s' in unit '%s' and conflicting with compiler generated class name for unit '%s' (Delphi)

E2471 Possibly parameterless constructors not allowed on record types (Delphi)

E2474 Record type required (Delphi)

E2475 '%s' directive not allowed in record helper type (Delphi)

E2501 Inline function cannot call nested routine '%s' (Delphi)

E2506 Method of parameterized type declared in interface section must not use local symbol '%s' (Delphi)

E2507 CLASS constraint and RECORD constraint cannot be specified together (Delphi)

E2508 type parameters not allowed on this type (Delphi)

E2510 Type '%s' is not a valid constraint (Delphi)

E2511 Type parameter '%s' must be a class type (Delphi)

E2512 Type parameter '%s' must be a non-nullable value type (Delphi)

E2513 Type parameter '%s' must have one public parameterless constructor named Create (Delphi)

E2514 Type parameter '%s' must support interface '%s' (Delphi)

E2515 Type parameter '%s' is not compatible with type '%s' (Delphi)

E2516 Constraint '%s' is conflicting with previous constraints (Delphi)

E2517 Operator '%s' must take %d parameter(s) (Delphi)

E2518 Operator '%s' must take least one '%s' type in parameters (Delphi)

E2519 First parameter type of operator '%s' and result type must be identical (Delphi)

E2520 Result type of operator '%s' must be Boolean type (Delphi)

E2521 Operator '%s' must take one '%s' type in parameter or result type (Delphi)

E2522 Operator '%s' can not convert to/from ancestor type (Delphi)

E2527 Helper type '%s' cannot be used in declarations (Delphi)

E2528 Type parameters not allowed on this method (Delphi)

E2529 Type parameters not allowed on operator (Delphi)

E2530 Type parameters not allowed on global procedure or function (Delphi)

E2531 Method '%s' requires explicit type argument(s) (Delphi)

E2532 Couldn't infer generic type argument from different argument types for method '%s' (Delphi)

E2533 Virtual, dynamic and message methods cannot have type parameters (Delphi)

E2534 Class constructor must not be virtual, dynamic nor message (Delphi)

E2535 Interface methods must not have parameterized methods (Delphi)

E2537 DEFAULT standard function expects a type identifier (Delphi)

E2538 DEFAULT standard function not allowed for this type (Delphi)

E2539 Missing implementation for abstract method '%s.%s' (Delphi)

E2540 Cannot seal abstract type '%s' (Delphi)

E2543 Instantiated data type '%s' too large: exceeds 2 GB (Delphi)

E2545 Cannot publish class property '%s' - only instance properties can be published (Delphi)

E2548 Cannot take a class reference of parameterized type (Delphi)

E2549 Cannot declare parameterized type derived from custom attribute class (Delphi)

E2556 Type of procedure result or untyped parameter cannot be used as type argument (Delphi)

E2551 There is no type parameterized methods of '%s' that can be used with these number of type parameters (Delphi)

E2552 CLASS or RECORD constraint and class type constraint cannot be specified together (Delphi)

E2553 Class type constraint cannot be specified more than once (Delphi)

E2555 Cannot capture symbol '%s' (Delphi)

E2560 $OBJTYPENAME not allowed for '%s'; only global and non-alias type symbol (Delphi)

E2561 Specified options for $EXTERNALSYM and $NODEFINE are not allowed for '%s'; only global and non-alias type symbol (Delphi)

E2562 Field identifier required (Delphi)

E2563 Specified interface type is not declared (Delphi)

E2564 Undefined type '%s' (Delphi)

E2565 Cannot use parameterized or instantiated type as custom attribute (Delphi)

E2568 Can't create new instance without CONSTRUCTOR constraint in type parameter declaration (Delphi)

E2569 Type parameter '%s' may need finalization - not allowed in variant record. Consider using RECORD constraint (Delphi)

E2570 Local procedure in generic method or method of generic type is not supported (Delphi)

E2571 Type parameter '%s' doesn't have class or interface constraint (Delphi)

E2572 RTTI visibility set constant expression of type System.TRttiVisibility expected (Delphi)

E2573 Illegal value for the ALIGN directive (valid for one of 1, 2, 4, 8 or 16) (Delphi)
E2574 Instantiated type can not be used for TYPE'd type declaration (Delphi)
E2575 RTTI for '%s' is too large; reduce scope with $RTTI or reduce type size (Delphi)
E2584 Weak attribute only allowed on fields and variables of type class or interface: '%s' (Delphi)
E2589 Helper type can not be declared in parameterized type context (Delphi)
E2590 Linker option string too long (Delphi)
E2591_Only cdecl functions may use varargs (Delphi)
E2597 '%s' (Delphi)
E2598 Virtual methods not allowed in record helper types (Delphi)
E2599 Field definition not allowed in helper type (Delphi)
E2601 Procedure NEW needs constructor identifier of '%s', but undeclared identifier '%s' found (Delphi)
E2602 Procedure DISPOSE needs destructor identifier of '%s', but undeclared identifier '%s' found (Delphi)
E2603 Constraint '%s' cannot be specified more than once (Delphi)
Linker error: %s (Delphi)
Linker error: %s: %s (Delphi)

失败消息

F1027 Unit not found '%s' or binary equivalents (%s) (Delphi)
F2039 Could not create output file '%s' (Delphi)
F2040 Seek error on '%s' (Delphi)
F2046 Out of memory (Delphi)
F2047 Circular unit reference to '%s' (Delphi)
F2048 Bad unit format - '%s' (Delphi)
F2051 Unit %s was compiled with a different version of %s.%s (Delphi)
F2063 Could not compile used unit '%s' (Delphi)
F2069 Line too long (more than 1023 characters) (Delphi)
F2084 Internal Error - %s%d (Delphi)
F2087 System unit incompatible with trial version (Delphi)
F2092 Program or unit '%s' recursively uses itself (Delphi)
F2220 Could not compile package '%s' (Delphi)
F2438 UCS-4 text encoding not supported. Convert to UCS-2 or UTF-8 (Delphi)
F2446 Unit '%s' is compiled with unit '%s' in '%s' but different version '%s' found (Delphi)
F2458 Cannot import metadata from Delphi 'library'. Use packages instead (Delphi)
F2462 Framework %s does not support thread local variables (%s.%s) (Delphi)
F2613 Unit '%s' not found. (Delphi)

提示消息

要禁用或启用提示消息，请参阅 $HINTS 指令和 IDE 支持以控制提示和警告。

H2077 Value assigned to '%s' never used (Delphi)
H2135 FOR or WHILE loop executes zero times - deleted (Delphi)
H2164 Variable '%s' is declared but never used in '%s' (Delphi)
H2219 Private symbol '%s' declared but never used (Delphi)
H2235 Package '%s' does not use or export '%s.%s' (Delphi)
H2244 Pointer expression needs no Initialize/Finalize - need ^ operator? (Delphi)
H2365 Override method %s.%s should match case of ancestor %s.%s (Delphi)
H2368 Visibility of property accessor method %s should match property %s.%s (Delphi)
H2369 Property accessor %s should be %s (Delphi)
H2384 CLS: overriding virtual method '%s.%s' visibility (%s) must match base class '%s' (%s) (Delphi)
H2440 Inline method visibility is not lower or same visibility of accessing member '%s.%s' (Delphi)
H2443 Inline function '%s' has not been expanded because unit '%s' is not specified in USES list (Delphi)
H2444 Inline function '%s' has not been expanded because accessing member '%s' is inaccessible (Delphi)
H2445 Inline function '%s' has not been expanded because its unit '%s' is specified in USES statement of IMPLEMENTATION section and current function is inline function or being inline function (Delphi)
H2456 Inline function '%s' has not been expanded because contained unit '%s' uses compiling unit '%s' (Delphi)
H2457 Inline function '%s' has not been expanded because contained unit '%s' uses compiling unit '%s' indirectly (Delphi)
H2505 'Self' is uninitialized. An inherited constructor must be called before entering a try block (Delphi)
H2509 Identifier '%s' conflicts with type parameters of container type (Delphi)
H2596 '%s' (Delphi)

警告消息

W1000 Symbol '%s' is deprecated (Delphi)
W1001 Symbol '%s' is specific to a library (Delphi)
W1002 Symbol '%s' is specific to a platform (Delphi)
W1003 Symbol '%s' is experimental (Delphi)
W1004 Unit '%s' is specific to a library (Delphi)
W1005 Unit '%s' is specific to a platform (Delphi)
W1006 Unit '%s' is deprecated (Delphi)
W1007 Unit '%s' is experimental (Delphi)
W1009 Redeclaration of '%s' hides a member in the base class (Delphi)
W1010 Method '%s' hides virtual method of base type '%s' (Delphi)
W1011 Text after final 'END.' - ignored by compiler (Delphi)
W1013 Constant 0 converted to NIL (Delphi)
W1014 String constant truncated to fit STRING%ld (Delphi)

W1015 FOR-Loop variable '%s' cannot be passed as var parameter (Delphi)
W1016 Typed constant '%s' passed as var parameter (Delphi)
W1017 Assignment to typed constant '%s' (Delphi)
W1018 Case label outside of range of case expression (Delphi)
W1021 Comparison always evaluates to False (Delphi)
W1022 Comparison always evaluates to True (Delphi)
W1023 Comparing signed and unsigned types - widened both operands (Delphi)
W1024 Combining signed and unsigned types - widened both operands (Delphi)
W1029 Duplicate %s '%s' with identical parameters will be inacessible from C++ (Delphi)
W1031 Package '%s' will not be written to disk because -J option is enabled (Delphi)
W1032 Exported package threadvar '%s.%s' cannot be used outside of this package (Delphi)
W1034 $HPPEMIT '%s' ignored (Delphi)
W1035 Return value of function '%s' might be undefined (Delphi)
W1036 Variable '%s' might not have been initialized (Delphi)
W1037 FOR-Loop variable '%s' may be undefined after loop (Delphi)
W1039 No configuration files found (Delphi)
W1040 Implicit use of Variants unit (Delphi)
W1041 Error converting Unicode char to locale charset. String truncated. Is your LANG environment variable set correctly (Delphi)
W1042 Error converting locale string '%s' to Unicode. String truncated. Is your LANG environment variable set correctly (Delphi)
W1043 Imagebase $%X is not a multiple of 64k. Rounding down to $%X (Delphi)
W1044 Suspicious typecast of %s to %s (Delphi)
W1045 Property declaration references ancestor private '%s.%s' (Delphi)
W1046 Unsafe type '%s%s%s' (Delphi)
W1047 Unsafe code '%s' (Delphi)
W1048 Unsafe typecast of '%s' to '%s' (Delphi)
W1049 value '%s' for option %s was truncated (Delphi)
W1050 WideChar reduced to byte char in set expressions (Delphi)
W1051 Duplicate symbol names in namespace. Using '%s.%s' found in %s. Ignoring duplicate in %s (Delphi)
W1055 Published caused RTTI ($M+) to be added to type '%s' (Delphi)
W1057 Implicit string cast from '%s' to '%s' (Delphi)
W1058 Implicit string cast with potential data loss from '%s' to '%s' (Delphi)
W1059 Explicit string cast from '%s' to '%s' (Delphi)
W1060 Explicit string cast with potential data loss from '%s' to '%s' (Delphi)
W1061 Narrowing given WideChar constant ('%s') to AnsiChar lost information (Delphi)
W1062 Narrowing given wide string constant lost information (Delphi)
W1063 Widening given AnsiChar constant ('%s') to WideChar lost information (Delphi)
W1064 Widening given AnsiString constant lost information (Delphi)
W1066 Lost Extended floating point precision. Reduced to Double (Delphi)
W1067 Unable to load DLL '%s' (Delphi)
W1068 Modifying strings in place may not be supported in the future (Delphi)
W1069 Feature will not be available in mobile Delphi compiler (Delphi)

W1070 Use of untyped pointer can disrupt instance reference counts (Delphi)

W1071 Implicit integer cast with potential data loss from '%s' to '%s' (Delphi)

W1072 Implicit conversion may lose significant digits from '%s' to '%s' (Delphi)

W1073 Combining signed type and unsigned 64-bit type - treated as an unsigned type (Delphi)

W1074 Unknown Custom Attribute (Delphi)

W1201 XML comment on '%s' has badly formed XML-'Whitespace is not allowed at this location.' (Delphi)

W1202 XML comment on '%s' has badly formed XML- 'Reference to undefined entity '%s'' (Delphi)

W1203 XML comment on '%s' has badly formed XML-'A name was started with an invalid character.' (Delphi)

W1204 XML comment on '%s' has badly formed XML-'A name contained an invalid character.' (Delphi)

W1205 XML comment on '%s' has badly formed XML-'The character '%c' was expected.' (Delphi)

W1206 XML comment on '%s' has cref attribute '%s' that could not be resolved (Delphi)

W1207 XML comment on '%s' has a param tag for '%s', but there is no parameter by that name (Delphi)

W1208 Parameter '%s' has no matching param tag in the XML comment for '%s' (but other parameters do) (Delphi)

警告或错误消息

这些消息可以是错误或警告，相应的首字母（E 或 W）取代消息编号中的 x。

x1008 Integer and HRESULT interchanged (Delphi)

x1012 Constant expression violates subrange bounds (Delphi)

x1019 For loop control variable must be simple local variable (Delphi)

x1020 Constructing instance of '%s' containing abstract method '%s.%s' (Delphi)

x1025 Unsupported language feature: '%s' (Delphi)

x1026 File not found '%s' (Delphi)

x1028 Bad global symbol definition '%s' in object file '%s' (Delphi)

x1030 Invalid compiler directive - '%s' (Delphi)

x1033 Unit '%s' implicitly imported into package '%s' (Delphi)

x1054 Linker error: %s (Delphi)

x1056 Duplicate resource Type %s, ID %s; File %s resource kept; file %s resource discarded (Delphi)

x2041 Read error on '%s' (Delphi)

x2042 Write error on '%s' (Delphi)

x2043 Close error on '%s' (Delphi)

x2044 Chmod error on '%s' (Delphi)

x2141 Bad file format '%s' (Delphi)

x2161 %s (Delphi)

x2162 %s: %s (Delphi)

x2243 Expression needs no Initialize/Finalize (Delphi)

x2269 Overriding virtual method '%s.%s' has lower visibility (%s) than base class '%s' (%s) (Delphi)

x2367 Case of property accessor method %s.%s should be %s.%s (Delphi)

x2421 Imported identifier '%s' conflicts with '%s' in '%s' (Delphi)