第2章　Object Pascal语言

本章内容：

- 注解
- 新的过程和函数特征
- 变量
- 常量
- 运算符
- Object Pascal类型
- 用户自定义类型
- 强制类型转换和类型约定
- 字符串资源
- 测试条件
- 循环
- 过程和函数
- 作用域
- 单元
- 包
- 面向对象编程
- 使用Delphi对象
- 方法
- 结构化的异常处理
- 运行期类型信息

这一章把Delphi的可视化元素放在一边，主要是为了向读者介绍 Delphi使用的语言 —— Object Pascal语言。首先介绍Object Pascal的基础，例如语言规则和结构；稍后介绍一些 Object Pascal的高级技术，例如类和异常处理。因为这本书不是针对入门者的书，它假定读者有使用其他高级计算机语言例如C、C++或Visual Basic的经验，最后将学习编程的概念，例如变量、类型、运算符、循环、条件、异常和对象，并跟C++和Visual Basic进行比较。

即使有使用Pascal的经验，你仍将发现这一章有用，因为通过这一章会学到 Pascal语法和语义的内在本质。

2.1　注解

作为起点，应该知道在Pascal代码中怎样写注解。Object Pascal支持三种类型的注解：花括号注解、圆括号/星号注解和C++风格的双斜杠注解。下面是每种类型的例子：

```
{使用花括号注解}
(*使用圆括号/星号注解*)
//C++风格的注解
```

前两种注解在本质上是相同的，编译器把在注解限定符头和注解限定符尾中间的内容当作注解。对于C++风格的注解来说，双斜杠后面到行尾的内容被认为是注解。

注意 相同类型的注解不能嵌套。虽然不同类型的注解进行嵌套在语法上是合法的，但我们不建议这样做。这里是一些例子：

```
{(*这是合法的*)}
(*{这是合法的}*)
(*(*这是非法的*)*)
{{这是非法的}}
```

2.2 新的过程和函数特征

因为自有编程语言以来，过程和函数是相当普遍的话题，所以在这里我们不打算深入，仅介绍一些新的或鲜为人知的特性。

2.2.1 圆括号

虽然不是在Delphi 5.0才有的，但Object Pascal中一个鲜为人知的特性是，当调用一个过程或函数时，如果被调用的过程和函数没有参数，圆括号是可以省略的。因此下面两个例子在语法上都是合法的：

```
Form1.Show;
Form1.Show();
```

当然，这个特性看上去作用不是很大，但是它对于那些同时使用 C++或Java(需要圆括号)的程序员来说特别有用。假如不能把时间全都花在 Delphi上，这个特性使你没必要记住在不同语言之间不同的函数调用规则。

2.2.2 重载

Delphi4引进了函数重载的概念(多个过程和函数有相同的名字，但有不同的参数列表)。所有需要重载的方法都需要声明，并用overload来指示，示例如下：

```
procedure Hello(I: Integer); overload;
procedure Hello(S: string); overload;
procedure Hello(D: Double); overload;
```

要注意的是，一个类的方法的重载稍微有点不同，在后面的 2.18.1节中介绍。虽然这是自 Delphi 1.0以来开发人员一直期望有的一个重要特性，但要记住小心使用它。有多个相同名字的过程和函数使得控制程序的执行和调试程序更加困难，正因为这样，对于使用重载这个特性，既不要回避，也不要滥用。

2.2.3 缺省值参数

缺省值参数是在Delphi 4中被引进的(当调用有缺省值参数的过程和函数时，可以不提供参数)。为了声明一个有缺省值参数的过程或函数，在参数类型后跟一个等号和缺省值，示例如下：

```
procedure HasDefVal(S: string; I: Integer = 0);
```

HasDefVal()过程能用下列两种方式调用。

第一种方式，两个参数都指定：

```
HasDefVal('hello', 26);
```

第二种方式，指定一个参数S，对I则用缺省值：

```
HasDefVal('hello');  //对于I，使用缺省值
```

在使用缺省值参数时要遵循下列几条规则：

- 有缺省值的参数必须在参数列表的最后。在一个过程或函数的参数列表中，没有缺省值的参数不能在有缺省值的参数的后面。
- 有缺省值的参数必须是有序类型、指针类型或集合类型。
- 有缺省值的参数必须是数值参数或常量参数，不能是引用 (out)参数或无类型参数。

有缺省值参数的最大好处是，在向一个已存在的过程和函数增加功能时，不必关心向后兼容的问题，例如，假定在一个单元中有一个函数为 AddInts()，它将两个数相加。

```
function AddInts(I1, I2: Integer): Integer;
begin
  Result := I1 + I2;
end;
```

如果想修改这个函数的功能使它能实现三个数相加，可能感到麻烦，因为增加一个参数将导致已经调用该函数的代码不能编译。由于有了有缺省值的参数，能扩展 AddInits()函数的功能而不必担心它的兼容性，示例如下：

```
function AddInts(I1, I2: Integer; I3: Integer = 0);
begin
  Result := I1 + I2 + I3;
end;
```

2.3 变量

你可能已经养成随时声明变量的习惯，“我需要另一个整数，所以我就在代码块的中间声明它”。如果这是你的习惯，在Object Pascal中使用变量时将不得不对自己有所限制。Object Pascal要求在一个过程、函数或程序前在变量声明段中声明它们。你可能过去习惯写这样的代码：

```
void foo(void)
{
  int x = 1;
  x++;
  int y = 2;
  float f;
  //... etc ...
}
```

在Object Pascal中，任何这样的代码要被整理成如下：

```
Procedure Foo;
var
  x, y: Integer;
  f: Double;
begin
  x := 1;
  inc(x);
  y := 2;
  //... etc ...
end;
```

注意 Object Pascal —— 像Visual Basic，但不像C和C++，是大小写不敏感的。采用大小写只是为了可读性好，所以就像本书所用的风格，能根据自己的需要使用大小写。如果一个名字是几个单词连在一起的，记住用大写来区分它们，例如，下列名字不清晰并很难读：

```
procedure thisprocedurenamemakesnosense;
```

然而，这样就好多了：

```
procedure ThisProcedureNameIsMoreClear;
```

对本书的代码风格的完整介绍，请看第6章“代码标准文档”。

你可能奇怪这样的结构化有什么好处，然而，你将看到 Object Pascal的结构风格导致它的代码的可读性好、维护性好并比C++或Visual Basic减少了错误的可能性。

注意Object Pascal是用下列语法将多个相同类型的变量合并并放在同一行上的：

```
VarName1, VarName2: SomeType;
```

当在Object Pascal中声明一个变量时，变量名在类型的前面，中间用冒号隔开，变量初始化通常要跟变量声明分开。

在Delphi 2中进入了一个新的语言特性，能在var块中对全局变量赋初值，这里有一些例子演示：

```
var
  i: Integer = 10;
  S: string  = 'Hello world';
  D: Double  = 3.141579;
```

注意 能赋初值的变量仅是全局变量，不是那些在过程或函数中局部变量。

提示 Delphi编译器自动对全局变量赋初值。当程序开始时，所有的整型数赋为0，浮点数赋为0.0，指针为null，字符串为空等等，因此，在源代码中不必对全局变量赋零初值。

2.4 常量

在Pascal中用const关键字来声明常量，就像在C语言中const关键字一样，这里有三个在C语言中定义的常量：

```
const float ADecimalNumber = 3.14;
const int i = 10;
const char * ErrorString = "Danger, Danger, Danger!";
```

C语言跟Object Pascal语言声明常量的主要差别是，在Object Pascal语言中不需要在对常量赋值时声明常量的类型，跟Visual Basic一样，编译器能根据常量的值自动判断常量的类型并分配内存；对于整型常量，编译器在运行时跟踪它的值，并不对它分配内存，例如：

```
const
  ADecimalNumber = 3.14;
  i = 10;
  ErrorString = 'Danger, Danger, Danger!';
```

注意 编译器根据如下规则来对常量分配内存：整型数被认为是最小的Integer类型(10被认为是ShortInt，32000被认为是SmallInt等等)；字符串值被认为是char 类型或string类型(通过$H定义)；浮点值被认为是 extended数据类型，除非小数点位数小于 4位(这种情况被当作 comp类型)；Integer和Char的集合类型被存储为它们自己。

当然，在声明变量时可以指定变量的类型，这样就能完全控制编译器对常量的处理，

```
const
  ADecimalNumber: Double = 3.14;
  I: Integer = 10;
  ErrorString: string = 'Danger, Danger, Danger!';
```

Object Pascal允许在const和var声明时用编译期间的函数，这些函数包括 Ord()、Chr()、Trunc()、Round()、High()、Low()和Sizeof()。例如，下列所有代码都是合法的：

```
type
  A = array[1..2] of Integer;
```

```
const
  w: Word = SizeOf(Byte);

var
  i: Integer = 8;
  j: SmallInt = Ord('a');
  L: Longint = Trunc(3.14159);
  x: ShortInt = Round(2.71828);
  B1: Byte = High(A);
  B2: Byte = Low(A);
  C: char = Chr(46);
```

警告 在32位Delphi中常量的类型定义是与在16位Delphi中常量的类型定义不同的。在16位的Delphi 1.0中，这种指定了类型的常量不是当作常量，而是被当作赋初值的变量或类型常量(typed constant)；然而，在Delphi 2.0以后指定了类型的常量才真正被当作是常量。 Delphi在Project菜单的Options对话框中的Compiler页上提供了向后兼容；或者可以用 $J编译开关来控制，缺省情况下，该开关是开的，向后兼容Delphi1.0，但是最好不要用这个方法，因为Object Pascal以后可能不再支持它。

如果程序试图改变常量的值， Delphi编译器就会有一个警告，认为程序违反了规则，因为常量值是只读的。Object Pascal通过在应用程序的代码页存贮常量来优化数据空间，如果对代码页和数据页的概念不清楚，请看第3章“ Win32 API ”。

注意 跟C和C++不一样，Object Pascal没有预编译。在Object Pascal中没有宏的概念，因此Object Pascal没有跟C一样的#define常量声明。虽然在Object Pascal中用$define进行条件编译，但不能用它来定义常量。在C或C++中用#define定义常量的地方在Object Pascal中用const来定义。

2.5 运算符

运算符是在代码中对各种数据类型进行运算的符号。例如，有能进行加、减、乘、除的运算符，有能访问一个数组的某个单元地址的运算符。本节将介绍各种 Pasical运算以及这些运算符跟C和Visual Basic运算符的区别。

2.5.1 赋值运算符

如果你是Pasical的新手， Delphi的赋值运算符可能是你最不习惯的事情之一。为了给一个变量赋值，用：=运算符，而不像在C或Visual Basic中用 = 运算符。Pascal程序员称它为获得运算符或赋值运算符，下列表达式：

```
Number1 := 5;
```

可以读成Number1获得值5，或Number1被赋值为5。

2.5.2 比较运算符

如果你用Visual Basic编过程序，可能对Delphi的比较运算符感到很熟悉，因为它们在本质上是相同的，比较运算符在所有程序语言中基本相同，所以本节只是简单介绍。

Object Pascal用=运算符对两个表达式或两个值进行逻辑比较运算， Object Pascal 中的=运算符跟在C语言中的==运算符相同，所以在C语言中的表达式If(x==y) 在Object Pascal 中写成If x=y。

注意 在Object Pascal中:=运算符式用来为一个变量赋值，而=运算符是比较两个操作数的值。

Delphi的不等于运算符是<>，相当于C语言中的！=运算符，为了判断两个表达式是否不等，用下列代码：

```
if x <> y then DoSomething
```

2.5.3 逻辑表达式

Pascal用单词and 和or作为逻辑与和逻辑或运算符，而C语言使用&&和 ||作为逻辑与和或的运算符。与和或最常用的是作为if语句或循环语句的一部分，就像下面两个例子演示的：

```
if (Condition 1) and (Condition 2) then
  DoSomething;

while (Condition 1) or (Condition 2) do
  DoSomething;
```

Pascal的逻辑非的运算符是 not，它是用来对一个布尔表达式取反，相当与 C语言中的！运算符，它同样也经常作为if语句的一部分，示例如下：

```
if not (condition) then (do something);   //如果条件错则...
```

表2-1是一个简明参考表，列出了Object Pascal、C/C++和Visual Basic的运算符。

表2-1 赋值、比较和逻辑运算符

运算符	Pascal	C/C++	Visual Basic
赋值	:=	=	=
比较	=	==	=或Is①
不等于	<>	!=	<>
小于	<	<	<
大于	>	>	>
小于等于	<=	<=	<=
大于等于	>=	>=	>=
逻辑与	and	&&	And
逻辑或	or	||	Or
逻辑非	not	!	Not

① Is比较符是用在对象之间，而=用于其他类型之间。

2.5.4 算术运算符

你应该对Object Pascal的大多数算术运算符很熟悉，因为它们跟在 C/C++和Visual Basic中的大致相同，表2-2列出了所有Object Pascal算术运算符以及跟C、Visual Basic的对照。

表2-2 算术运算符

运算符	Pascal	C/C++	Visual Basic
加	+	+	+
减	-	-	-
乘	*	*	*
浮点数除	/	/	/
整数除	div	/	/
取模	mod	%	Mod
指数	无	无	^

你可能注意到了Pascal和Visual Basic对浮点数和整型数除法运算用了不同的除法运算符，这点不像在C/C++语言中。在两个整数相除时，div运算符自动截取余数取整。

注意 对不同类型的表达式相除要选用不同的除法运算符，如果用div 运算符对两个浮点数或用/运算符对两个整型数进行除法运算，Object Pascal编译器将提示出错了，请看下面的代码：

```
var
  i: Integer;
  f: Real;
begin
  i := 4 / 3;          // 这将引起编译错误
  f := 3.4 div 2.3;    // 这将引起编译错误
end;
```

许多其他语言不区分浮点数和整数相除，而是进行浮点数相除，然后根据需要将结果转化为整型数，从性能上看，这可能开销要大一些，而Pascal的div运算符更快、更有效。

2.5.5 按位运算符

按位运算符能修改一个变量的单独各位。最常用的按位运算符能把一个数左移或右移，或对两个数按位执行与、取反、或和异或运算。移位运算符 shl和shr分别相对于C语言中的<<和>>运算符，表2-3列出这些按位运算符。

表2-3 按位运算符

运算符	Pascal	C/C++	Visual Basic
与	and	&	And
取反	not	~	Not
或	or	|	Or
异或	xor	^	Xor
左移	shl	<<	无
右移	shr	>>	无

2.5.6 加减运算过程

加减运算过程用来对一个给定的整型数执行加 1或减1运算，这是经过优化的代码， Pascal不提供像C++中的++和- -等类似的运算符，但Pascal 提供了Inc()和Dec()来执行相同的功能。

用一个或两个参数来调用Inc()或Dec()，例如，下面的代码分别对variable执行加1和减1 操作：

```
Inc(variable);
Dec(variable);
```

同上面相比较，用Inc()和Dec()分别对变量执行加 3 和减 3 操作：

```
Inc(variable, 3);
Dec(variable, 3);
```

表2-4列出了在不同语言中执行加减运算的运算符。

表2-4 加减运算符

运算符	Pascal	C	Visual Basic
加	Inc()	++	无
减	Dec()	- -	无

注意 如果允许编译优化，Inc()和Dec()过程就产生如下的机器码语法：variable:=variable+1;

用这样的方法对变量进行加减运算，可能感到更方便。

2.6 Object Pascal类型

Object Pascal的最大特点是，它的数据类型特别严谨，这表示传递给过程或函数的实参必须和定义过程或函数时的形参的类型相同。不可能在Pascal中看到一些著名编译器例如C编译器所提示的可疑的指针转换等编译警告信息。这是因为Object Pascal编译器不允许用一种类型的指针去调用形参为另一种类型的函数(无类型的指针除外)。

2.6.1 类型的比较

Delphi的基本数据类型跟C和Visual Basic的相同，表2-5对照列出了Object Pascal的基本数据类型以及C/C++和Visual Basic的基本数据类型。你可能想把这一页折起来，因为这张表提供了当在Delphi中调用不是Delphi的动态连接库(DLL)或目标文件(OBJ)中的函数时用于匹配类型的最好的参考。

表2-5 Pascal、C/C++、Visual Basic数据类型的对照

变量类型	Pascal	C/C++	Visual Basic
8位有符号整数	ShortInt	char	无
8位无符号整数	Byte	BYTE，unsigned short	Byte
16位有符号整数	SmallInt	short	Short
16位无符号整数	Word	unsigned short	无
32位有符号整数	Integer，Longint	int，long	Integer Long
32位无符号整数	Cardinal，LongWord	unsigned long	无
64位有符号整数	Int64	_int64	无
4字节浮点数	Single	float	Single
6字节浮点数	Real48	无	无
8字节浮点数	Double	double	Double
10字节浮点数	Extnded	long double	无
64位货币值	currency无	无	Currency
8字节日期/时间	TDateTime	无	Date
16字节variant	Variant，OleVariant，TVarData	VARIANT Variant①，OleVariant①	Variant(缺省)
1字节字符	Char	char	无
2字节字符	WideChar	WCHAR	
固定长度字节的字符串	ShortString	无	无
动态字符串	AnsiString	AnsiString①	String
以Null结束的字符串	PChar	char*	无
以Null结束的宽字符串	PWideChar	LPCWSTR	无
动态2字节字符串	WideString	WideString①	无
1字节布尔值	Boolean，ByteBool	(任何1字节数)	无
2字节布尔值	WordBool	(任何2字节数)	Boolean
4字节布尔值	BOOL，LongBool	BOOL	无

① 模拟对应的Object Pasacl类型的Borland C++ Builder类。

注意 如果要移植Delphi 1.0的16位代码，请记住，无论是Integer还是Cardinal 类型都已经从16位扩展到32位。更准确地说，在Delphi 2和Delphi 3中，Cardinal被看作是31位的无符号整数，在Delphi 4以后，Cardinal才真正成为32位的无符号整数。

警告 在Delphi 1、2和3中，Real是6字节的浮点数，这是Pascal特有的数据类型，和其他的语言不兼容。在Delphi 4中，Real是Double类型的别名，6字节的浮点数仍然有，但现在是Real48。通过编译开关{$REALCOMPATIBILITY ON}可以使Real仍然代表6字节的浮点数。

2.6.2 字符

Delphi有三种类型的字符：

- AnsiChar 这是标准的1字节的ANSI字符，程序员都对它比较熟悉。
- WideChar 这是2字节的Unicode字符。
- Char 在目前相当于AnsiChar， 但在Delphi以后版本中相当于WideChar.

记住因为一个字符在长度上并不表示一个字节，所以不能在应用程序中对字符长度进行硬编码，而应该使用Sizeof()函数。

注意 Sizeof()标准函数返回类型或实例的字节长度。

2.6.3 字符串

字符串是代表一组字符的变量类型，每一种语言都有自己的字符串类型的存储和使用方法。Pascal类型有下列几种不同的字符串类型来满足程序的要求：

- AnsiString 这是Pascal缺省的字符串类型，它由 AnsiChar 字符组成，其长度没有限制，同时与null结束的字符串相兼容。
- ShortString 保留该类型是为了向后兼容Delphi 1.0，它的长度限制在255个字符内。
- WideString 功能上类似于AnsiString，但它是由WideChar字符组成的。
- PChar 指向null结束的Char字符串的指针，类似于C的char * 或lpstr类型。
- PAnsiChar 指向null结束的AnsiChar字符串的指针。
- PWideChar 指向null结束的WideChar字符串的指针。

缺省情况下，如果用如下的代码来定义字符串，编译器认为是 AnsiString 字符串：

```
var
  S:string; //编译器认为S的类型是AnsiString
```

当然，能用编译开关$H来将string类型定义为ShortString，当$H编译开关的值为负时， string变量是ShortString类型；当$H编译开关的值为正时(缺省情况)，字符串变量是AnsiString类型。下面的代码演示了这种情况：

```
var
  {$H-}
  S1:string;  //S1是ShortString类型
  {$H+}
  S2:string;  //S2是AnsiString类型
```

使用$H规则的一个例外是，如果在定义时特地指定了长度 (最大在255个字符内)，那么总是ShortString：

```
var
S: string[63];  //63个字符的ShortString字符串
```

1. AnsiString类型

AnsiString(或长字符串)类型是在Delphi 2.0开始引入的，因为Delphi 1.0的用户特别需要一个容易使用而且没有255个字符限制的字符串类型，而AnsiString正好能满足这些要求。

虽然AnsiString在外表上跟以前的字符串类型几乎相同，但它是动态分配的并有自动回收功能，正是因为这个功能AnsiString有时被称为生存期自管理类型。Object Pascal能根据需要为字符串分配空间，所以不用像在C/C++中所担心的为中间结果分配缓冲区。另外， AnsiString字符串总是以null字符结束的，这使得AnsiString字符串能与Win32 API中的字符串兼容。实际上， AnsiString类型是一个指向在堆栈中的字符串结构的指针。

警告 Borland没有将长字符串类型的内部结构写到文档中，并保留了在Delphi后续版本中修改长字符串内部格式的权利。在这里介绍的情况主要是帮助你理解 AnsiString类型是怎样工作的并且在程序中要避免直接依赖于AnisString结构的代码。

程序员如果在Delphi 1.0中避免了使用字符串的内部结构，则把代码移植到Delphi 2.0没有任何问题。而依赖于字符串内部结构写代码的程序在移植到Delphi 2.0时必须修改。

正如图2-1演示的， AnsiString字符串类型有引用计数的功能，这表示几个字符串都能指向相同的物理地址。因此，复制字符串因为仅仅是复制了指针而不是复制实际的字符串而变得非常快。

图2-1 显示了AnsiString在内存中分配的情况

当两个或更多的 AnsiString类型共享一个指向相同物理地址的引用时， Delphi内存管理使用了copy-on-write技术，一个字符串要等到修改结束，才释放一个引用并分配一个物理字符串。下面的例子显示了这些概念：

```
var
  S1,S2:string;
begin
  //给S1赋值，S1的引用计数为1
  S1:='And now for something...';
  S2:=S1;  //现在S2与S1指向同一个字符串，S1的引用计数为2
  //S2现在改变了，所以它被复制到自己的物理空间，并且S1的引用计数减1
S2:=S2+'completely different1';
end;
```

生存期自管理类型

除了AnsiString以外，Delphi还提供了其他几种生存期自管理类型，这些类型包括：WideString、Variant、OleVariant、interface、dispinterface和动态数组，这些类型在本章稍后有介绍，现在，我们重点讨论究竟什么是生存期自管理，以及它们是如何工作的。

生存期自管理类型，又被称为自动回收类型，是指那些在被使用时就占用一定的特殊资源，而在它离开作用域时自动释放资源的类型。当然不同的类型使用不同的资源，例如，AnsiString类型在被使用时就为字符串占用内存，当它超出作用域时就释放被字符串占用的内存。

对于全局变量，这种情况是相当直观的：作为应用程序结束代码的一部分，编译器自动插入代码来清除每一个生存期自管理类型的全局变量。因为在应用程序在被装入时，全局变量都被初始化为0，每一个全局变量将用初始化时的0、空或其他值来指示它还没有被使用，基于这种方法，终止代码只清除那些确实在应用程序中被使用的全局变量。

对于局部变量来讲，这种情况稍微有的复杂：首先，在过程或函数开始运行时，编译器插入的代码保证初始化这些局部变量，接着，编译器产生一个try...finally的异常处理块，它包

容整个函数体，最后，编译器在finally块插入代码来清除生存期自管理变量(异常处理块在2.19节“结构化异常处理”中介绍)，为了能记住这些，请看下面的代码：

```
procedure Foo;
var
S:string;
begin
//过程体
//在这里用S
end;
```

虽然这个过程看起来简单，如果考虑进由编译器插入的代码，它看起来像下面的代码：

```
procedure Foo;
var
S:string;
begin
S:='';
try
//过程体
//在这里用S
finally
//在这里清除S
end;
end;
```

(1) 字符串运算符

能用+运算符或Concat()函数来连接两个字符串，推荐使用+运算符，因为Concat()函数主要用来向后兼容，下面代码演示了+运算符和Concat()函数的用法：

```
{用+运算符}
var
  S, S2: string
begin
  S:= 'Cookie ':
  S2 := 'Monster';
  S := S + S2;   { Cookie Monster }
end.
```

```
{ 用Concat() }
var
  S, S2: string;
begin
  S:= 'Cookie ';
  S2 := 'Monster';
  S := Concat(S, S2);   { Cookie Monster }
end.
```

注意 在Object Pascal中，通常用一对单引号来把字符串括起来，例如'A String'。

提示 Concat()是众多“编译器魔术”过程和函数中的一个，其他还有ReadLn()、WriteLn().、它们没有Object Pascal的定义。这些函数接收不确定个数的参数或可选的参数，它们不能根据Object Pascal语言来定义，正因为这样，编译器为每一个这样的函数提供了特殊的条件，并产生一个对堆栈函数的调用，这些堆栈函数定义在System单元中，它们为了越过Pascal的语言规则，通常是用汇编语言实现的。

除了“编译器魔术”字符串函数和过程外，在SysUtil单元中有许多函数和过程用来使字符串更易使用，请查阅Delphi的联机帮助“字符串处理例程”。

此外，可以在本书附带的CD-ROM的\Source\Utils目录中的StrUtils单元中找到更多用来处理字符串的过程和函数。

(2) 长度和分配

第一次声明AnsiString时，它是没有长度的，因此在字符串中就没有为字符分配空间。为了对字符串分配空间，用一行字母或另一个字符串对它进行赋值，或者用 SetLength()过程，就像下面所列出来的：

```
var
  S:string ;              // 字符初始化时，没有长度
begin
  S:='Doh!';              //为字符串的字母分配足够的空间
  {或者}
  S:=OtherString;         //增加OtherString的引用计数,
                          //{假定OtherString指向一个合法的字符串}
  {或者}
  SetLength(S,4);         //分配4个字符的空间
end;
```

能像数组一样对字符串进行索引，但注意索引不要超出字符串的长度，例如，下面的代码会引起一个错误：

```
var
  S:string;
begin
  S[1]:='a';    //不能工作，因为S没有被分配空间
end;
```

然而，代码改成如下，就能正常工作了：

```
var
  S:string;
begin
  SetLength(S,1);
  S[1]:='a';              //现在S有足够空间来容纳字符
end;
```

(3) Win32的兼容

正如前面所提到，AnsiString字符串总是null 结束的。因此，它能跟以null结尾的字符串兼容，这就使得调用Win32 API 函数或其他需要PChar型字符串的函数变得容易了。只要把一个字符类型强制转换为PChar类型(在2.8节“强制类型转换和类型约定”中将介绍强制类型转换)。下面的代码演示了怎样调用Win32的GetWindowsDirectory()函数，这个函数需要一个PChar类型的参数：

```
var
  S:String;
begin
  SetLength(S,256);          //重要！首先给字符串分配空间
  //调用API函数，S 现在包含目录字符串
GetWindowsDirectory(PChar(S),256);
```

如果使用了将AnsiString字符串强制转换为PChar类型的函数和过程，在使用结束后，要手工把它

的长度恢复为原来以null结束的长度。STRUTILS单元中的RealizeLenght()函数可以实现这一点：

```
procedure RealizeLength(var S:string);
begin
  SetLength(S,StrLen(PChar(S)));
end;
```

调用ReallizeLength():

```
var
  S:string;
begin
  SetLength(S,256);      //重要！首先给字符串分配空间
  //调用函数，S现在包含目录字符串
  GetWindowDirectory(PChar(S),256);
  RealizeLength(S);  //设置S的长度为null结束的长度
end;
```

注意 在练习将一个字符串转换为PChar类型时要小心，因为字符串在超出其作用范围时有自动回收的功能，因此当进行P:=PChar(Str)的赋值时，P的作用域(生存期)应当大于Str的作用域。

(4) 移植性问题

当要移植Delphi 1.0的应用程序时，一定要注意几个关于AnsiString的问题：

- 在使用PString(指向ShortString字符串的指针)的地方，应当替换成string类型。记住：AnsiString已经是一个指向字符串的指针。
- 不能再通过字符串的第0个元素来设置或得到字符串的长度，只能通过 Length()函数来得到字符串的长度，通过SetLength()过程来设置字符串的长度。
- 不再需要通过调用StrPas()和StrPCopy()来进行字符串与PChar之间的转换，正如上面所提到的，可以把AnsiString 强制类型转换为PChar。如果要把PChar的内容复制到AnsiString，直接用赋值语句：

```
StringVar: =PCharVar;
```

警告 对长字符串设置长度时，必须用SetLength()过程，过去那种通过直接访问字符串第0个元素来设置长度的方法，在应用程序从16位的Delphi 1.0升级到32位时会出现问题。

2. ShortString类型

如果你是一个Delphi的老手，应该知道ShortString类型是Delphi 1.0中字符串的类型，ShortString类型有时又被称为Psacal 字符串(Psacal string)或长度-字节字符串(length-byte string)。请记住，$H 编译开关的值用来决定当变量声明为字符串时，它是被当作AnsiString类型还是被当作ShortString类型。

在内存中，字符串就像是一个字符数组，在字符串的第 0个元素中存放了字符串的长度，紧跟在后的字符就是字符串本身。ShortString缺省的最大长度为256个字节，这表示在ShortString中不能有大于255个字符(255个字符+1个长度字节=256)。相对于AnsiString来说，用ShortString是相当随意的，因为编译器会根据需要为它分配空间，所以不用担心中间结果是不是要预先分配内存。图2-2演示了Pascal字符串在内存中的分配方式。

图2-2

一个ShortString变量用下面的代码声明和初始化：

```
var
  S: ShortString;
begin
  S := 'Bob the cat.';
end.
```

当然，能用short类型限定符和一个长度限制来为 ShortString分配小于256个字节的空间，示例如下：

```
var
  S: string[45];  { 一个45个字符的ShortString字符串 }
begin
  S := 'This string must be 45 or fewer characters.';
end.
```

如上代码，这个字符串肯定是 ShortString，而不再受$H编译开关的影响，能指定的短字符串的最大长度是255个字符。

不要存放比分配给字符串的空间长度更长的字符，如果声明了一个变量是 string[8]，并试图对这个变量赋值为‘ a_pretty_darn_long_string ’，这个字符串将被截取为仅有8个字符，就要丢失数据。

当用数组的下标来访问ShortString中的一个特定字符时，如果下标的索引值大于声明时 ShortString的长度，则会得到假的结果或造成内存混乱。例如，假定像下面那样声明了一个变量：

```
var
  Str:string[8];
```

如果试图写这个字符串的第10个元素，则有可能使其他变量的内存混乱。

```
var
  Str:string[8];
  i:Integer;
begin
  i:=10;
  Str[i]:='s';    //内存混乱
```

可以在Project Options 对话框中选中Range Checking复选框，这样编译器会自动加上特殊的逻辑在运行时捕捉此类错误。

提示 虽然在程序中包括范围检查能发现字符串错误，但范围检查多少都影响应用程序的性能。通常使用的方法是在开发程序或调试程序的阶段用范围检查，而在确信程序稳定时，去掉范围检查。

跟AnsiString类型字符串不一样， ShortString跟以null结尾的字符串不兼容，正因为这样，用ShortString调用Win32函数时，要做一些工作。下面这个ShortStringAsPChar()函数是在STRUTILS.PAS单元中定义的。

```
func function ShortStringAsPChar(var S:ShortString):PChar;
{这函数能使一个字符串以null结尾，这样就能传递给需要PChar类型参数的Win32 API函数，如果字符串超过254个字符，多出的部分将被截掉}
begin
  if Length(S)=High(S) then Dec(S[0]); { 如果S 太长，就截取一部分 }
  S[Ord(Length(S))+1]:=#0;             { 把null加到字符串的最后 }
  Result:=@S[1];                       { 返回PChar化的字符串 }
end;
```

警告 Win32 API函数需要以null结尾的字符串，不要把ShortString字符串传递给API函数，因为编译器将报错，长字符串可以传递给Win32 API函数。

2. WideString类型

WideString类型像AnsiString一样是生存期自管理类型，它们都能动态分配、自动回收并且彼此能相互兼容，不过WideString和AnsiString的不同主要在三个方面：

- WideString由WideChar字符组成，而不是由AnsiChar字符组成的，它们跟Unicode字符串兼容。
- WideString用SysAllocStrLen()API函数进行分配，它们跟OLE的BSTR字符串相兼容。
- WideString没有引用计数，所以将一个WideString字符串赋值给另一个WideString字符串时，就需要从内存中的一个位置复制到另一个位置。这使得 WideString在速度和内存的利用上不如AnsiString有效。

就像上面所提到的，编译器自动在 AnsiString类型和WideString类型的变量间进行转换。示例如下：

```
var
  W:wideString;
  S:string;
begin
    W:='Margaritaville';
  S:=W;  // wideString转换成AnsiString
  S:='Come Monday';
  W:=S;  // AnsiString转换成WideString
end;
```

为了能灵活地运用 WideString类型， Object Pascal重载了Concat()、 Copy、Insert()、Length()、Pos()和SetLength()等例程以及+、=和<>等运算符。下面的代码在语法上是正确的：

```
var
  W1,W2:Widestring;
  P:Integer;
begin
  W1:='Enfield';
  W2:='field';
  If W1<>W2 then
    P:=Pos(W1,W2);
end;
```

就像AnsiString和ShortString类型一样，能用数组的下标来访问WideString中一个特定的字符：

```
var
  W:WideString;
  C:WideChar;
begin
  W:='Ebony and Ivory living in prefect harmony';
  C:=W[Length(W)];  //C包含W字符串的最后一个字符
end;
```

4. 以null结束的字符串

正如前面所提到的， Delphi有三种不同的以 null结束的字符串类型： PChar、PAnsiChar和PWideChar。它们都是由 Delphi的三种不同字符组成的。这三种类型在总体上跟 PChar是一致的。PChar之所以保留是为了跟Delphi 1.0和Win32 API兼容，而它们需要使用以null结束的字符串，PChar被定义成一个指向以null(零)结束的字符串指针(如果对指针的概念不熟悉，请继续读下去，在本章的后面要详细地介绍)。与AnsiString和WideString类型不同，PChar的内存不是由Object Pascal自动产生和管理的，要用 Object Pascal的内存管理函数来为PChar所指向的内存进行分配。 PChar字符串的理论最大长度是4GB，PChar变量在内存中的分布见图2-3。

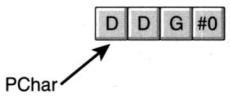

图2-3 PChar的内存分布

提示 在大多数情况下，AnsiString类型能被用成PChar，应该尽可能地使用AnsiString，因为它对字符串内存的管理是自动，极大地减少了应用程序中内存混乱的错误代码，因此，要尽可能地避免用PChar类型以及对它相应进行人工分配内存。

正如在前面所提到的， PChar变量需要人工分配和释放存放字符串的内存。通常，用 StrAlloc()函数为PChar缓冲区分配内存，但是其他几种函数也能用来为 PChar类型分配函数，包括 AllocMem()、GetMem()、StrNew()和VirtualAlloc() API函数。这些函数有相应的释放内存的函数。表 2-6列出了几个分配内存的函数以及它们相应的释放内存的函数。

表2-6 内存分配和释放函数

内存分配函数	内存释放函数
AllocMem()	FreeMem()
GlobalAlloc()	GlobalFree()
GetMem()	FreeMem()
New()	Dispose()
StrAlloc()	StrDispose()
StrNew()	StrDispose()
VirtualAlloc()	VirtualFree()

下面的例子演示了使用PChar和string类型时的内存分配技术：

```
var
  P1,P2:PChar;
  S1,S2:string;
begin
  P1:=StrAlloc(64*SizeOf(Char));  //P1指向一个分配了63个字符的缓冲区
  StrPCopy(P1,'Delphi 5');        //复制一组字母到P1
  S1:='Developer's Guide';        //在S1中放几个字母
  P2:=StrNew(PChar(S1));          //P2指向S1的备份
  StrCat(P1,P2);                  //连接P1和P2
  S2:=P1;                       //S2现在为'Delphi 5 Developer's Guide ’
  StrDispose(P1);                 //清除P1和P2的缓冲区。
  StrDispose(P2);
end.
```

首先注意到，在为P1分配内存时StrAlloc()中用到的SizeOf(char)。要记住在以后Delphi版本中一个字符的长度要从一个字节变成两个字节，因此，不能假定一个字符的长度为一个字节， Sizeof() 就保证了不管字符长度是多少都能正确地分配内存。

StrCat()用来连接两个PChar字符串。注意，这里不能像对长字符串和 ShortString类型那样用+运算符来连接两个字符串。

StrNew()函数用来把字符串S1中的值拷贝到P2中，使用这个函数要小心，要避免出现内存被覆盖的错误，因为StrNew()函数只为字符串分配足够但不浪费的内存，请看下面的例子：

```
var
P1,P2:PChar;
begin
  P1:=StrNew(‘Hello’);    //只分配够P1用的内存
  P2:=StrNew(‘World’);    //只分配够P2用的内存
  StrCat(P1,P2);          //小心：内存出现混乱了！
```

```
  .
  .
  .
end;
```

提示 和其他字符串类型一样，Object Pascal也为操作PChar类型提供了一些操作函数和过程，请在Delphi的联机帮助“String-handing routines(null-terminated)”。

另外，还可以在本书附带的CD-ROM的\Source\Utils目录中的StrUtils中找到更多的针对以null结束的字符串的过程和函数。

2.6.4 变体类型

Delphi 2.0引进了一个功能强大的数据类型，称为变体类型 (Variant)，主要是为了支持OLE自动化操作。实际上，Delphi的Variant封装了OLE使用的Variant，但Delphi的Variant在Delphi程序的其他领域也很有用。正如不久就要学到的，Object Pascal是唯一能在运行期间和编译期间识别Variant 的语言。

Delphi 3引进了一个新的被称为OleVariant 类型，它跟Variant基本一致，但是它只能表达与OLE 自动化操作相兼容的数据类型。本节介绍Variant,然后介绍OleVariant，并对两者进行比较。

1. Variant能动态改变类型

有时候变量的类型在编译期间是不确定的，而 Variant能够在运行期间动态地改变类型，这就是引入Variant 类型目的。例如，下面的代码在编译期间和运行期间都是正确的：

```
var
  V:Variant;
begin
  V:='Delphi is Great!';   //Variant 此时是一个字符串
  V:=1;                    //Variant 此时是一个整数
  V:=123.34;               //Variant 此时是一个浮点数
  V:=True;                 //Variant 此时是一个布尔值
  V:=CreateOleObject('word.Basic');   //Variant此时是一个OLE 对象
end;
```

Variant能支持所有简单的数据类型，例如整型、浮点型、字符串、布尔型、日期和时间、货币以及OLE自动化对象等。注意Variant不能表达Object Pascal对象。Variant可以表达不均匀的数组(数组的长度是可变的，它的数据元素能表达前面介绍过的任何一种类型，也可包括另一个 Variant数组)。

2. Variant的结构

Variant类型的数据结构是在System单元中定义的，如下所示：

```
type
  PVarData = ^TVarData;
  TVarData = packed record
    VType: Word;
    Reserved1, Reserved2, Reserved3: Word;
    case Integer of
      varSmallint: (VSmallint: Smallint);
      varInteger:  (VInteger: Integer);
      varSingle:   (VSingle: Single);
      varDouble:   (VDouble: Double);
      varCurrency: (VCurrency: Currency);
      varDate:     (VDate: Double);
      varOleStr:   (VOleStr: PWideChar);
      varDispatch: (VDispatch: Pointer);
```

```
    varError:    (VError: LongWord);
    varBoolean:  (VBoolean: WordBool);
    varUnknown:  (VUnknown: Pointer);
    varByte:     (VByte: Byte);
    varString:   (VString: Pointer);
    varAny:      (VAny: Pointer);
    varArray:    (VArray: PVarArray);
    varByRef:    (VPointer: Pointer);
  end;
```

TVarData结构占用16个字节的内存，TVarData结构的头两个字节中的值用于指示variant代表的数据类型，下面的代码显示了可能出现在TVarData结构中VType域中的各种值。紧接的6个字节是保留的，剩下的8个字节包含实际的数值或一个指向由 Variant表示的数据的指针。另外，这个结构直接映射了OLE的Variant类型的实现。

```
{ Variant类型代码 }
const
  varEmpty    = $0000;
  varNull     = $0001;
  varSmallint = $0002;
  varInteger  = $0003;
  varSingle   = $0004;
  varDouble   = $0005;
  varCurrency = $0006;
  varDate     = $0007;
  varOleStr   = $0008;
  varDispatch = $0009;
  varError    = $000A;
  varBoolean  = $000B;
  varVariant  = $000C;
  varUnknown  = $000D;
  varByte     = $0011;
  varStrArg   = $0048;
  varString   = $0100;
  varAny      = $0101;
  varTypeMask = $0FFF;
  varArray    = $2000;
  varByRef    = $4000;
```

注意 在上面的列表代码中你可能发现，Variant不能代表指针和类。

可能在TVarData列表中注意到，TVarData记录是一个可变记录。不要把它跟 Variant类型相混淆，虽然可变记录和Variant类型在英文中的名字相同，但它们代表了两个完全不同的概念。可变记录允许多个数据域重叠在相同的内存 (像C/C++中的联合类型)，这将在本章的2.7.3节“记录”中详细介绍。在TVarData可变记录中的case语句指示了variant类型所代表的数据的类型，例如：如果VType域包含的值为VarInteger，TVarData记录的8个字节中的4个字节用来存放一个整型值；如果VType的值为VarByte，8个字节中仅有1个字节存放数据。

你可能注意到了VType的值可以为VarString，这时8个字节并不存放真正的字符串，它存放指向字符串的指针，这是很重要的一点，因为它能直接访问一个 Variant的域，示例如下：

```
var
  V: Variant;
begin
  TVarData(V).VType := varInteger;
  TVarData(V).VInteger := 2;
end;
```

你必须明白，在某些情况下这是个危险的操作，因为它可能导致对字符串或其他生存期自管理类型的引用丢失，这将引起应用程序内存错误。在下面一节你就会明白“自动回收”的含义。

3. Variant是生存期自管理的

Delphi自动分配和释放Variant类型所需的内存，请看下面的例子，对一个Variant变量赋了字符串：

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := S;
  ShowMessage(V);
end;
```

在本章的前面讨论了生存期自管理类型，有几种情况没有讲清楚，现在继续。首先 Delphi将Variant初始化为一个不确定的值。在赋值期间，它把 VType域设为VarString，并把字符串指针拷贝到它的VString域，然后增加字符串S的引用计数。当这个Variant类型的变量离开其作用域时(过程结束或调用了返回代码)，它被清除并减少S的引用计数，Delphi是通过插入try...finally代码块来隐式实现这个过程的，示例如下：

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := Unassigned;  // 将variant初始化为“空”
  try
    V := S;
    ShowMessage(V);
  finally
    // 现在清除和variant有关的资源
  end;
end;
```

当另一种类型的数据赋值给variant变量时，也会隐式地释放资源，请看下面的代码：

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  V := S;
  V := 34;
end;
```

这段代码是下列伪代码的具体体现：

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  清除变量V，确保它初始化为“空”；
  try
    V.VType := varString; V.VString := S; Inc(S.RefCount);
    清除变量V，释放对字符串S的引用
    V.VType := varInteger; V.VInteger := 34;
  finally
    清除与变量V有关的资源
  end;
end;
```

如果理解了上面的代码，就明白为什么不建议你直接对 TVarData记录的域进行操作，请看下面的例子：

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  V := S;
  TVarData(V).VType := varInteger;
  TVarData(V).VInteger := 32;
  V := 34;
end;
```

虽然这段代码看起来很安全，但它可能引起内存错误。作为一条普遍的规则，请不要直接访问TVarData的数据域。如果确实需要直接访问，必须清楚正在干什么。

4. Variant的强制类型转换

能显式地把一个表达式强制类型转换成Variant类型，例如，表达式：

```
Variant(X)
```

使得Variant中的类型代码跟表达式x的结果的数据类型一致，x的结果的数据类型必须是整数、实数、货币型、字符串、字符或布尔型。

也可以把Variant类型的表达式强制类型转换为简单类型，例如：

```
V := 1.6;
```

其中，V是一个Variant类型的变量，下面的代码对变量V进行转换：

```
S:=String(V);    //S现在是字符串1.6
I:=Integer(V);   //I现在取整为整数2
B:=Boolean(V);   //如果V的值为0，则B为False,否则为True;
D:=Double(V);    //D现在为浮点数1.6
```

这是根据类型转换规则将 Variant类型转换成简单类型的结果，关于类型转换规则的详细细节在Delphi的Object Pascal Language Guide中。

顺便说一下，在上面的例子中没有必要通过强制类型转换，只要用赋值语句就能实现，下面的代码是正确的：

```
V := 1.6;
S := V;
I := V;
B := V;
D := V;
```

在这里转换成目标数据类型是隐式发生的。不过，由于转换是在运行期间进行的，编译器会对这段程序附加许多代码。如果确切地知道 Variant所包含数据的类型，最好用显式地强制类型转换，以加快速度，尤其当Variant出现在表达式中时。关于这一点将在后面讨论。

5. 表达式中的Variant

能在表达式中用Variant，并通过=、*、/、div、mod、shl、shr、and、or、xor、not、:=、<>、<、>、<=和>=进行运算。

当Variant变量出现在表达式中时，Delphi知道怎样基于Variant中的内容进行运算，例如，如果有V1和V2两个Variant类型的变量，它们都包含整型数，那么表达式V1+V2的结果是两个整型数的和；然而，如果V1和V2是两个字符串，那么结果是两个字符串的连接。那么，如果 V1和V2中包含不同类型的数据那又怎样呢？Delphi为了能执行运算用了更复杂的规则，例如，如果 V1包含字符串‘ 4.5 ’，而V2包含一个浮点数，那么V1将被转换成一个浮点数并加上V2的值，下面的代码演示了这样的规则：

```
var
  V1,V2,V3:Variant;
begin
```

```
  V1:='100';   //V1包含一个字符串类型
  V2:='50';    //V2包含一个字符串类型
  V3:=200;     //V3包含一个整型数
  V1:=V1+V2+V3;
end;
```

基于上面所提到的复杂的规则，这段代码的结果乍一看是整型数 350；然而，如果你仔细看，就发现情况并非如此。因为代码的执行顺序是从左向右，首先是执行 V1+V2，因为两个variant变量都包含了字符串，所以执行字符串连接运算，结果为字符串‘ 10050 ’;然后这个结果再加上在V3中包含的整型数，因为V3包含的是整型数，所以字符串‘ 10050 ’转换成一个整型数并加上200，这样最后结果是10250。

为了能顺利地计算，在等式中， Delphi尽量把Variant转换成有利于运算的类型，如果 Delphi碰到了搞不清的表达式，就会触发一个“ Invalid variant type conversion ”异常，示例如下：

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'hello';
  V1 := V1 / V2; // 产生异常
end;
```

正如上面所提到的，如果知道在表达式中要进行运算的数据的类型，最好把 Variant变量显式地转换，看下面的例子：

```
V4 := V1 * V2 / V3;
```

在这个等式的最后结果出来以前，会有一个运行期的函数对这个表达式中的各个操作数进行兼容性检查，然后进行必要的转换。这将导致额外的开销。一种最好的解决方法是不要用 Variant类型，如果确实要用，最好显式地对Variant变量进行类型转换，这样有关数据类型的转换在编译期间就能完成，例如：

```
V4 := Integer(V1) * Double(V2) / Integer(V3);
```

记住在转换前必须知道Variant所表示的数据类型。

6. 空和null

在这里简单介绍Variant变量中VType的两个特殊的值，第一个是varEmpty，这表示这个Variant还没有赋值，这是当这个变量进入作用域时由编译器给它指定的初始值。另一个是 varNull，它跟varEmpty最大的区别是，它确实代表了一个数值null，而不是没有值。没有值和值为null在涉及到数据库表的字段值运算时显得尤其重要。在第 28章“ 编写桌面数据库应用程序 ”中，会看到在数据库应用程序中Variant变量是怎样被应用的。

另一个不同的地方是，在执行有 Variant变量的等式而Variant变量的VType值为varEmpty时，会产生“ invalid variant operation ”异常。如果VType的值为varNull，就不会有这种情况。如果一个表达式中包含值为null的Variant变量，则整个表达式的结果为null。

如果想用varEmpty或VarNull对一个Variant变量进行赋值或比较，在System单元中提供了两个特殊的值Unassigned和null，它们的VType值分别为varEmpty和varNull。

警告 由于Variant类型提供了更灵活的机制，你可能想用Variant类型来代替传统的数据类型。然而，这将增加程序代码长度，并使得程序运行起来很慢。另外它也使得程序代码很难维护。在有些情况下Variant特别有用。事实上，VCL中有好几个地方都用到了variant，尤其是在ActiveX和数据库领域。因为这些地方需要复杂的数据类型。一般来说，应当使用传统的数据类型，除非对灵活性的要求超过了对程序性能的要求。不确定的数据类型会产生不确定的缺陷。

7. Variant数组

前面曾经提到过，variant能表达不均匀的数组，下面的代码在语法上是合法的：

```
var
  V: Variant;
  I, J: Integer;
begin
  I := V[J];
end;
```

请记住，虽然上面的代码在编译时能通过，但在运行时会得到一个异常。因为 V不是一个Variant类型的数组，Object Pascal提供了若干支持Variant数组的函数，允许建立Variant数组，下面就有两个：VarArrayCreate()和VarArrayOf()。

(1) VarArrayCreate()函数

VarArrayCreate()函数在System单元中是这样定义的：

```
function VarArrayCreate(const Bounds: array of Integer;
  VarType: Integer): Variant;
```

为了用VarArrayCreate()函数，必须传递想要建立数组的边界和数组中每个元素的类型代码（第一个参数是开放数组，关于它将在本章的2.12节详细介绍），下面的代码创建一个元素为整型的Variant数组，并对每一个元素赋值：

```
var
  V: Variant;
begin
  V := VarArrayCreate([1, 4], varInteger); // 创建一个4个元素的数组
  V[1] := 1;
  V[2] := 2;
  V[3] := 3;
  V[4] := 4;
end;
```

如果知道了怎样用简单的数据类型创建 variant数组，就能把varVariant作为建立variant数组的类型代码传递给函数。这样在数组中的每一个数据都可以有不同的类型。同时能通过传递附加的边界来创建一个多维数组，下面的例子创建了一个边界为[1..4, 1..5]的数组：

```
V := VarArrayCreate([1, 4, 1, 5], varInteger);
```

(2) VarArrayOf()

VarArrayOf()函数在System单元中是这样定义的：

```
function VarArrayOf(const Values: array of Variant): Variant;
```

这个函数创建一个一维数组，它的元素由 Values参数指定，下面的例子创建了一个数组，它的三个元素分别为整型、字符串型和浮点数型：

```
V := VarArrayOf([1, 'Delphi', 2.2]);
```

(3) 与Variant数组相关的函数和过程

除了VarArrayCreate()和VarArrayOf(),在System单元中定义了多个支持 Variant数组的函数和过程，列表如下：

```
procedure VarArrayRedim(var A: Variant; HighBound: Integer);
function VarArrayDimCount(const A: Variant): Integer;
function VarArrayLowBound(const A: Variant; Dim: Integer): Integer;
function VarArrayHighBound(const A: Variant; Dim: Integer): Integer;
function VarArrayLock(const A: Variant): Pointer;
procedure VarArrayUnlock(const A: Variant);
function VarArrayRef(const A: Variant): Variant;
function VarIsArray(const A: Variant): Boolean;
```

VarArrayRedim()函数能修改variant数组的最高限。VarArrayDimCount()函数返回Variant数组的维数，VarArrayLowBound()和VarArrayHighBound()函数分别返回Variant数组的下边界和上边界。VarArrayLock()和VarArrayUnlock()函数比较特殊，将在下一节介绍。

VarArrayRef()主要用来解决向OLE服务器传送Variant数组时要出现的问题。当向自动化服务器的方法传送一个包含Variant数组的Variant变量时，问题就产生了，例如：

```
Server.PassVariantArray(VA);
```

这里传送的不是一个variant数组，而是一个包含了variant数组的variant变量，这是有明显区别的。如果自动化服务器希望接收一个variant数组而不是指向它的一个引用，如果用上面的语法来调用自动化的方法，就会产生一个错误。在这种情况下就要用VarArrayRef()，示例如下：

```
Server.PassVariantArray(VarArrayRef(VA));
```

VarIsArray()函数是一个简单的布尔检查函数，如果传递给它的参数是一个Variant数组则返回True，否则返回False.

(4) 用VarArrayLock()和VarArrayUnlock()初始化一个大数组

Variant数组在OLE自动化技术中尤其重要，因为它提供了向OLE服务器传送二进制数据的唯一的方法(就像你在第23章“COM和ActiveX”要学到的，这时指针不再有效)。然而，如果用得不正确，它也会产生负作用。请看如下代码：

```
V := VarArrayCreate([1, 10000], VarByte);
```

它创建一个有10 000字节的Variant数组。假定有另一个相同长度的非Variant类型的数组，并且想把非variant类型的数组复制到variant类型的数组中去。通常这都是通过用循环语句对variant数组的单元进行赋值实现的。示例如下：

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  for i := 1 to 10000 do
    V[i] := A[i];
end;
```

上面这段代码的主要问题在于，大量的时间花费在对Variant数组的初始化上，因为在对每一个元素进行赋值时，都要通过运行时逻辑来检查并判断数据类型的兼容性、每个元素的位置等等。为了避免运行时检查，要用VarArrayLock()函数和VarArrayUnlock()过程。

VarArrayLock()函数在内存中锁定数组，使数组不再移动和改变大小，并能返回一个指向数组数据的指针。而VarArrayUnlock()过程用来对VarArrayLock()函数锁定的数组进行解锁，使数组能重新移动或改变大小。在锁定数组后，能用更有效的方法对数组进行初始化，例如用指向数组数据的指针调用Move()过程，下面的代码也对Variant数组进行初始化，但它更有效：

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  P := VarArrayLock(V);
  try
    Move(A, P^, 10000);
  finally
    VarArrayUnlock(V);
  end;
end;
```

8. 其他支持Variant的函数

还有其他一些函数支持Variant,它们都定义在System单元中，下面是这些函数：

```
procedure VarClear(var V: Variant);
procedure VarCopy(var Dest: Variant; const Source: Variant);
procedure VarCast(var Dest: Variant; const Source: Variant; VarType: Integer);
```

```
function VarType(const V: Variant): Integer;
function VarAsType(const V: Variant; VarType: Integer): Variant;
function VarIsEmpty(const V: Variant): Boolean;
function VarIsNull(const V: Variant): Boolean;
function VarToStr(const V: Variant): string;
function VarFromDateTime(DateTime: TDateTime): Variant;
function VarToDateTime(const V: Variant): TDateTime;
```

VarClear()过程清除Variant变量并将VType域的值设为varEmpty.

VarCopy()将Source复制到Dest。

VarCast()将一个Variant转换成指定的类型并存储在另一个Variant变量中。

VarType()返回指定Variant的varXXX类型代码。

VarAsType()跟VarCast()的功能一样。

VarIsEmpty()如果一个Variant变量的类型代码是varEmpty 则返回True。

VarIsNull() 判断Variant变量是否包含null值。

VarToStr()将一个Variant变量转换成字符串表达式 (如果Variant为varEmpty或varNUll则为空字符串)。

VarFromDateTime() 返回一个Variant变量，它存放着指定的TDateTime类型的值。

VarToDateTime() 返回在Variant中的TDateTime类型的值。

9. OleVariant

OleVariant与Variant很相像，两者的差别在于OleVariant仅支持跟自动化相兼容的类型。目前，不能跟自动化相兼容的 VType是VarString，即AnsiString类型。当试图把 AnsiString字符串赋值给一个OleVariant变量时， AnsiString自动转化为OLE BSTR类型并作为varOleStr存储在Varaint变量中。

2.6.5 Currency

在Delphi 2.0中引进了一种新的数据类型Currency。它的主要目的是用来进行财经计算，它不同于能在数字间移动小数点的浮点数，它的小数点是固定的，在小数前有15位数字，在小数点后有4位数字。当移值Delphi 1.0的程序并涉及到货币时，最好用Currency来代替Single、real、Double和Extended。

2.7 用户自定义类型

整型、字符串型和浮点数型不能足够地表达在现实世界中的情况，而这些情况往往是需要程序来解决的。为了弥补这种不足，必须建立新的数据类型来表示现实问题中的变量。在 Pascal中，这种用户自定义类型用记录(record)或对象(object)来表示。

2.7.1 数组

Object Pascal 允许你建立各种类型变量的数组(除了文件类型)。下面的例子声明了一个数组，这个数组有八个整型数：

```
var
  A: Array[0..7] of Integer;
```

它相当于C语言中的：

```
int A[8];
```

另外，它相当于Visual Basic中的

```
Dim A(8) as Integer
```

Object Pascal的数组有一个不同于其他语言的特性，它们的下标不必以某个数为基准。像下面的

例子，能从28开始定义一个有3个元素的数组：

```
var
  A: Array[28..30] of Integer;
```

因为Object Pascal的下标不是必须从0或1开始的，在for循环中使用数组时一定要小心。在编译器中有两个内置的函数High() 和Low()，它们分别返回一个数组变量或数组类型的上边界和下边界。在for 循环中用这两个语句能使程序更加稳定和易于维护，示例如下：

```
var
  A: array[28..30] of Integer;
  i: Integer;
begin
  for i := Low(A) to High(A) do  // 不要把在循环中硬编码！
    A[i] := i;
end;
```

提示 字符的数组通常以0为基准，它能用来传递给需要PChar类型变量的函数，这是编译器特别允许的。

为了定义多维数组，用逗号开，示例如下：

```
var
  // 整型的二维数组
  A: array[1..2, 1..2] of Integer;
```

为了访问多维数组，在方括号中用逗号隔开每一维：

```
I := A[1, 2];
```

2.7.2 动态数组

动态数组是在编译时不知道维数，在运行时动态分配的数组。为了声明一个动态数组，只要在声明时不要指定维数，就像这样：

```
var
  // 字符串的动态数组
  SA: array of string;
```

在用动态数组前，用SetLength()过程为数组分配内存：

```
begin
  // 为33个元素分配空间
  SetLength(SA, 33);
```

一旦空间被分配了，你就能像访问普通数组一样访问动态数组：

```
SA[0] := 'Pooh likes hunny';
OtherString := SA[0];
```

注意 动态数组通常是以0为基准的。

动态数组是生存期自管理的，所以在用完它们以后没有必要释放，因为在离开作用域时它们会被释放。然而，可能在离开作用域前，就需要删除动态数组 (例如它用了很多内存)。要这么做，仅需要把nil赋值给动态数组：

```
SA:=nil;  //释放SA
```

利用引用对动态数组进行操作，在语义上类似于 AnsiString类型，而不像普通的数组。这里有一个小实验：在下面的代码运行结束后A1[0]的值是多少？

```
var
  A1, A2: array of Integer;
begin
  SetLength(A1, 4);
```

```
A2 := A1;
A1[0] := 1;
A2[0] := 26;
```

正确答案是26！，因为赋值语句A2:=A1并不创建新的数组，仅将A1数组的引用赋给A2，因此对A2数组的任何操作都影响到A1。如果想用A1的完全拷贝赋值给A2，用标准过程Copy():

```
A2 := Copy(A1);
```

当这行代码运行结束， A1和A2是两个独立的数组，但它们的初始值是相同的，改变其中的一个不会影响到另一个。在调用Copy()时，能任意指定起始单元和要复制的元素个数，例如：

```
//从元素1开始，复制2个元素
A2:=Copy(A1,1,2);
```

动态数组也能定义为多维的。为了定义多维数组，在声明时对每一维用一个 array关键字:

```
var
  // 整型的二维动态数组
  IA: array of array of Integer;
```

在为了给动态数组分配空间调用SetLength()时，也要将另一维的长度传递过去：

```
begin
  // IA是一个5×5的整型数组
  SetLength(IA, 5, 5);
```

访问多维动态数组跟访问多维普通数组一样，在方括号中每一个元素用逗号隔开：

```
IA[0,3] := 28;
```

2.7.3 记录

在Object Pascal中用户自定义的结构被称为记录。它相当于C语言中struct， Visual Basic中的Type。下面的例子对三者进行了比较：

```
{ Pascal }
Type
  MyRec = record
    i: Integer;
    d: Double;
  end;
```

```
/* C */
typedef struct {
  int i;
  double d;
} MyRec;
```

```
'Visual Basic
Type MyRec
  i As Integer
  d As Double
End Type
```

当使用记录时，用小圆点来访问它的域，下面是例子：

```
var
  N: MyRec;
begin
  N.i := 23;
  N.d := 3.4;
end;
```

Object Pascal也支持可变记录，它允许在记录中不同的数据共同覆盖相同的内存。不要跟 Variant

数据类型相混淆，可变记录允许彼此覆盖的数据能被单独访问。如果对 C/C++语言很熟悉，你一眼就能看出来它跟C/C++中的union概念一样。下面的例子演示了一个可变记录，其中 Double、Integer和char共同占用了相同的内存：

```
type
  TVariantRecord = record
    NullStrField: PChar;
    IntField: Integer;
   case Integer of
     0: (D: Double);
     1: (I: Integer);
     2: (C: char);
end;
```

注意 Object Pascal规则声明：一个记录的可变部分不能是生存期自管理的类型。

如果在C语言中就是这样定义的：

```
struct TUnionStruct
{
  char * StrField;
  int IntField;
  union
  {
    double D;
    int i;
    char c;
  };
};
```

2.7.4 集合

集合是Pascal特有的数据类型，在Visual Basic、C或C++都没有(虽然C++ Builder提供了一种模板类称为集合，它模仿 Pascal集合的行为)。集合用一种有效的手段来表示一组有序数、字符和枚举值。声明一个集合用关键字set of，并在其后跟上有序类型或一个集合可能值的有限子集。示例如下：

```
type
TCharSet=set of char;  // 可能的值：#0-#255
TEnum=(Monday,Tuesday,Wednesday,Thursday,Fridy);
TEnumSet=set of TEnum;  // 包含了TEnum值的任何组合
TSubrangeSet=set of 1..10;  // 可能的值：1-10
TAlphaSet=Set of 'A'..'z';     // 可能的值：'A'-'z'
```

注意，一个集合最多只能有255个元素。另外，只有有序的类型才能跟关键字 set of，因此下列的代码是非法的：

```
type
TIntSet=set of Integer;  // 非法：太多的元素
TStrSet=set of string;   // 非法：不是有序的类型
```

集合在内部以位的形式存储它的元素，这使得在速度和内存利用上更有效。集合如果少于 32个元素，它就存储在CPU的寄存器中，这样效率就更高了，为了用集合类型得到更高的效率。记住，集合的基本类型的元素数目要小于32。

1. 使用集合

当使用集合的元素时，使用方括号。下面代码表明如何使用集合类型的变量并给它赋值。

```
type
  TCharSet = set of char;       // possible members: #0 - #255
  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
  TEnumSet = set of TEnum;  // can contain any combination of TEnum members
var
  CharSet: TCharSet;
  EnumSet: TEnumSet;
  SubrangeSet: set of 1..10; // possible members: 1 - 10
  AlphaSet: set of 'A'..'z'; // possible members: 'A' - 'z'
begin
  CharSet := ['A'..'J', 'a', 'm'];
  EnumSet := [Saturday, Sunday];
  SubrangeSet := [1, 2, 4..6];
  AlphaSet := [];  // Empty; no elements
end;
```

2. 集合操作符

Object Pascal提供了几个用于集合的运算符，用这些运算符可以判断集合和集合之间的关系，对集合增删元素，对集合进行求交集运算。

(1) 关系运算

用in运算符来判断一个给定的元素是否在一个集合中，下面的代码判断在前面所定义的集合CharSet中是否有字母‘ S ’：

```
if 'S' in CharSet then
  //继续运行
```

下面的代码判断在EnumSet中是否没有Monday:

```
if not (Monday in EnumSet) then
  //继续运行
```

(2) 增删元素

用+、-运算符或Include()和Exclude()过程，能对一个集合变量增删元素：

```
Include(CharSet, 'a');          //在集合中增加'a';
CharSet:=CharSet+['b'];         //在集合中增加'b';
Exclude(CharSet, 'x');          //在集合中删除'x';
CharSet:=CharSet-['y', 'z'];    //在集合中删除'y'和'z';
```

提示 尽可能地用Include()和Exclude()来增删元素，尽可能地少用+、-运算符。因为Include()和Exclude()仅需要一条机器指令，而+和-需要13+6n(n是集合的按位的长度)条机器指令。

(3) 交集

用*运算符来计算两个集合的交集，表达式Set1*Set2的结果是产生的集合的元素在Set1和Set2集合中都存在，下面的例子用来判断在一个给定的集合中是否有某几个元素：

```
if {'a', 'b', 'c'}*CharSet={'a', 'b', 'c'} then
  //继续程序
```

2.7.5 对象

可以将对象类型当作是记录类型，只是它还包括了函数和过程。 Delphi的对象模型在后面2.17节“ 使用Delphi的对象 ”中详细介绍。所以这一节主要介绍Object Pascal 对象的基本语法。一个对象被定义成这样：

```
Type
  TChildObject = class(TParentObject);
```

```
    SomeVar: Integer;
    procedure SomeProc;
  end;
```

虽然Delphi的对象和C++中的对象不完全一样，但上面的代码大致和在C++中的定义相同：

```
class TChildObject : public TParentObject
{
  int SomeVar;
  void SomeProc();
};
```

要定义一个方法，类似于定义函数和过程 (在2.12节“过程和函数”中讨论)，只是要加上对象名字和小圆点.:

```
procedure TChildObject.SomeProc;
begin
  { 过程体 }
end;
```

Object Pascal中的小圆点．在功能上类似于Visual Basic 中的．运算符和C++的::运算符。虽然三种语言都允许使用类，但只有 Object Pascal和C++允许以完全面向对象的方法定义新类，关于这一点在2.16节“面向对象编程”中详细介绍。

注意 Object Pascal的对象和C++的对象在内存中的布局不一样。因此，在Delphi中不能用C++的对象，反过来也是一样，不过，第13章“核心技术”将介绍在Delphi和C++中如何共享对象。

一个例外的情况是，在Borland C++ Builder中创建的新类，用_declspec(delphiclass)指令可以直接映射为Object Pascal类。但这样的类与普通C++对象不兼容。

2.7.6 指针

一个指针变量指示了内存的位置，在本章介绍 PChar类型时，就有使用指针的例子， Psacal通用指针类型的名称是Pointer。Pointer有时又被称为无类型指针，因为它只指向内存地址，但编译器并不管指针所指向的数据，这一点与 Pascal严谨的风格似乎不相称，所以建议你在大部分情况下用有类型的指针。

注意 指针是一种高级技术，对于编写程序来说不是必须的，当你对它运用熟练以后，它会成为你写程序的好工具。

有类型指针在你的应用程序的Type部分用^(或pointer)运算符声明。对于有类型指针来说，编译器能准确地跟踪指针所指内容的数据类型，这样用指针变量，编译器就能跟踪正在进行的工作。以下是声明指针的例子。

```
Type
  PInt=^Integer;              // PInt现在是指向Integer的指针
  Foo=record                  // 一个记录类型
    GobbledyGook: string;
    Snarf:Real;
  end;
  PFoo=^Foo;                  // PFoo是一个指向foo类型的指针
var
  P:Pointer;                  // 一个无类型的指针
  P2:PFoo;                    // PFoo的实例
```

注意 C语言的程序员可能注意到了，Object Pascal的^运算符与C语言中*运算符相似，Pascal

的Pointer类型对应与C的void *类型。

要记住一个指针变量仅仅是存储一个内存的地址，为指针所指向的内容分配空间是程序员要干的工作，用在前面介绍过的并列在表2-6中的内存分配例程来分配内存。

注意 如果一个指针没有指向任何数据，它的值是nil，它就被称为是零(nil)指针或空(null) 指针。

要访问一个指针所指向的内容，在指针变量名字的后面跟上 ^运算符。这种方法称为对指针取内容。下面的代码演示怎样用指针：

```
Program PtrTest;
Type
  MyRec = record
    I: Integer;
    S: string;
    R: Real;
  end;
   PMyRec = ^MyRec;
 var
   Rec : PMyRec;
begin
   New(Rec);       // 为Rec分配内存
   Rec^.I := 10;   // 对Rec中的域赋值
   Rec^.S := 'And now for something completely different.';
   Rec^.R := 6.384;
   { Rec现在满了 }
   Dispose(Rec);   // 不要忘了释放空间
end.
```

什么时候用New()

用New()函数能为一个指针分配指定长度的内存空间。在为某结构分配内存时，因为编译器知道要分配的内存的大小，所以调用 New()就能分配到所需的字节，而且它比 GetMem()或AllocMem()更安全，更易于使用。但不能用New()为Pointer 或PChar变量分配内存，因为编译器不知道需要分配多大的内存。另外要记住用Dispose()函数来释放用New()分配的内存。

当编译器不知道要分配多少内存时，就要用到 GetMem()和AllocMem()，在对PChar和Pointer类型分配内存时，编译器不可能提前告诉你要分配多少，因为它们有长度可变特性。要注意，不要试图操作分配空间以外的数据，因为这是导致“ Access Violation ”错误最常见的原因。用FreeMem()来释放由GetMem()和AllocMem()分配的内存。顺便说一下，AllocMem()要比GetMem()安全，因为AllocMem()总是把分配给它的内存初始化为零。

C程序员在学习Object Pascal时感到头痛的是， Object Pascal对指针类型的检查非常严格，例如，下面的代码中变量a和变量b并不兼容：

```
var
  a: ^Integer;
  b: ^Integer;
```

相反，在C语言中它们是兼容的：

```
int *a;
int *b
```

Object Pascal认为每一个指针类型是相异的，为了把 a的值赋给b，你必须建立一个新的类型，示例如下：

```
type
```

```
  PtrInteger = ^Integer;  // 建立新类型
var
  a, b: PtrInteger;       // 现在a和b相兼容了
```

2.7.7 类型别名

Object Pascal能为已经定义的类型创建新的名字，又称别名。例如，如果想为 Integer创建一个新的名字MyReallyNiftyInteger，用下列代码实现：

```
type
  MyReallyNiftyInteger = Integer;
```

新定义的类型别名跟它的源类型在任何时候都兼容，这样在凡是出现 Integer的地方都能用MyReallyNiftyInteger来替换。

如果要创建一个被编译器认为是独特的、全新的类型别名，就要用到 type关键字：

```
type
  MyOtherNeatInteger = type Integer;
```

这样，当MyOtherNeatInteger用于赋值目的时，它就转换为Integer，但在用作var 和out参数时，它和Integer是不兼容的。因此下面的代码在语法上是正确的：

```
var
  MONI: MyOtherNeatInteger;
  I: Integer;
begin
  I := 1;
  MONI := I;
```

相反，下面的代码就不能编译：

```
procedure Goon(var Value: Integer);
begin
  // 一些代码
end;

var
  M: MyOtherNeatInteger;
begin
  M := 29;
  Goon(M);   // 错误：M跟Integer 不兼容。
```

编译器除了进行兼容性检查外，在运行时还能对这些新的类型产生类型信息，这样就能为简单类型创建独特的属性编辑器，你将在第22章“高级组件技术”学到的这一操作。

2.8 强制类型转换和类型约定

强制类型转换是一种技术，通过它能使编译器把一种类型的变量当作另一种类型。由于 Pascal有定义新类型的功能，因此编译器在调用一个函数时对形参和实参类型匹配的检查是严格的。因此为了能通过编译检查，经常需要把一个变量的类型转换为另一个变量的类型。例如，假定要把一个字符类型的值赋给一个byte类型的变量：

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := c;    // 编译器要提示错误
end.
```

在下面的代码中，强制类型转换把c转换成byte类型，事实上强制类型转换是告诉编译器你知道你正在干什么，并要把一种类型转换为另一种类型：

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := byte(c);    // 编译器不再报错
end.
```

注意 只有当两个变量的数据长度一样时，才能对变量进行强制类型转换。例如，不能把一个Double强制类型转换成 Integer。为了把一个浮点数类型转换为一个整型，要用 Trunc()或Round()函数。为了把整型转换成一个浮点数类型的值，用下列的赋值语句：

```
FloatVar :=intVar;
```

Object Pascal还支持用as操作符在对象之间进行类型转换，这在本章的 2.20节“运行期类型信息”一节中介绍。

2.9 字符串资源

从Delphi 3开始，就能用关键字resourcestring将字符串资源直接用在Object Pascal源代码中。字符串资源是字母串(通常用来显示给用户)，并物理地存储在与应用程序或库相关联的资源中而不是直接内嵌在源代码中。在源代码中就能引用字符串资源而不是字母串本身，通过把字符串和源代码分开，只要替换字符串资源就能在不同的语言环境中切换。字符串资源用 resourcestring关键字以标识符=母串形式定义，下面是例子：

```
resourcestring
  ResString1 = 'Resource string 1';
  ResString2 = 'Resource string 2';
  ResString3 = 'Resource string 3';
```

资源中的字符串能像字符串常量一样在源代码中使用：

```
resourcestring
  ResString1 = 'hello';
  ResString2 = 'world';

var
  String1: string;

begin
  String1 := ResString1 + ' ' + ResString2;
  .
  .
  .
end;
```

2.10 测试条件

这一节把Pascal中的if和case语句与C语言、Visual Basic语言中的类似结构进行比较。我们假定你以前用过这些语句，这里不再花时间详细介绍。

2.10.1 if语句

在执行一段代码以前， if语句能让你判断某个条件是否满足。下面的代码分别列出了 if语句在

Pascal、C和Visual Basic中的用法：

```
{ Pascal }
if x = 4 then y := x;
/* C */
if (x == 4) then y = x;

'Visual Basic
If x = 4    y = x
```

注意 如果在一条if语句中有多个条件，你需要用括号把这几个条件分别用括号括起来，例如：

```
if (x = 7) and (y = 8) then
```

下面写法将导致编译器警告：

```
if x = 7 and y = 8 then
```

在Pascal中的begin和end，就像是C和C++中的{和}，例如，下面的代码是当一个条件满足时要执行多条语句：

```
if x = 6 then begin
  DoSomething;
  DoSomethingElse;
  DoAnotherThing;
end;
```

用if...else能组合多个条件：

```
if x =100 then
  SomeFunction
else if x = 200 then
  SomeOtherFunction
else begin
  SomethingElse;
  Entirely;
end;
```

2.10.2 case语句

在Pascal中的case语句就像是C和C++中的switch语句。case语句用来在多个可能的情况中选择一个条件，而不再需要用一大堆if..else if..else if结构，下面的代码是Pascal的case语句：

```
case SomeIntegerVariable of
  101 : DoSomething;
  202 : begin
      DoSomething;
      DoSomethingElse;
    end;
  303 : DoAnotherThing;
  else DoTheDefault;
end;
```

注意 case语句的选择因子必须是有序类型，而不能用非有序的类型如字符串作为选择因子。

这里是C语言中的swith语句。

```
switch (SomeIntegerVariable)
{
  case 101: DoSomeThing; break;
  case 202: DoSomething;
            DoSomethingElse; break
  case 303: DoAnotherThing; break;
```

```
  default: DoTheDefault;
}
```

2.11 循环

循环是一种能重复执行某一动作的语言结构， Pascal中的循环结构和其他语言中的循环结构相类似，所以这一节不再特别介绍循环了，仅仅列出了在Pascal中要用到的各种循环结构。

2.11.1 for循环

for循环适合用在事先知道循环次数的情况下，下面的代码在一个循环中把控制变量加到另一个变量中，共重复10次：

```
var
  I, X: Integer;
begin
  X := 0;
  for I := 1 to 10 do
    inc(X, I);
end.
```

上面的例子在C语言中是这样的：

```
void main(void) {
  int x, i;
  x = 0;
  for(i=1; i<=10; i++)
    x += i;
}
```

上面的例子在Visual Basic中是这样的：

```
X = 0
For I = 1 to 10
  X = X + I
Next I
```

警告 在Delphi 1.0中，允许对控制变量赋值；而从Delphi 2.0开始，不再允许对控制变量赋值，因为32位编译器对循环进行了优化。

2.11.2 while循环

while循环结构用在先判断某些条件是否为真，然后重复执行某一段代码的情况下。 while的条件是在循环体执行前进行判断的。用while循环的典型例子是当文件没有到达文件结尾时，对文件进行某一重复操作。下面的例子演示了每次从文件中读一行并写到屏幕上：

```
Program FileIt;

{$APPTYPE CONSOLE}

var
  f: TextFile;  // 一个文本文件
  S: string;
begin
  AssignFile(f, 'foo.txt');
  Reset(f);
  while not EOF(f) do begin
    readln(f, S);
```

```
    writeln(S);
  end;
  CloseFile(f);
end.
```

Pascal中的while循环基本上跟C中的while循环和Visual Basic中Do While循环一样。

2.11.3 repeat...until

repeaat...until循环与while循环相似，但考虑问题的角度不同，它在某个条件为真前一直执行给定的代码。它不像 while循环，因为条件测试在循环的结尾，所以循环体至少要执行一遍。 Pascal的repeat...until语句大致上同于C语言中的do...while语句。

例如，下面的代码不断地把一个计数器加1，直到它大于100为止：

```
var
  x: Integer;
begin
  X := 1;
  repeat
    inc(x);
  until x > 100;
end.
```

2.11.4 Break()过程

在while、for或repeat循环中调用Break()，使得程序的执行流程立即跳到循环的结尾，在循环中当某种条件满足时需要立即跳出循环，这时调用 Break()。Pascal中的Break()类似于C语言中的break 和Visual Basic中的Exit语句。下面的代码演示了在5次循环后跳出循环。

```
var
  i: Integer;
begin
  for i := 1 to 1000000 do
  begin
    MessageBeep(0);            // 让计算机蜂鸣
    if i = 5 then Break;
  end;
end;
```

2.11.5 Continue( )过程

如果想跳过循环中部分代码重新开始下一次循环，就调用 Continue()过程。注意下面的例子在执行第一次循环时continue()后的代码不执行：

```
var
  i: Integer;
begin
  for i := 1 to 3 do
  begin
    writeln(i, '. Before continue');
    if i = 1 then Continue;
    writeln(i, '. After continue');
  end;
end;
```

2.12 过程和函数

作为程序员，对于过程和函数的概念应该很熟悉了。一个过程是一段程序代码，它在被调用时能

执行某种特殊功能并能返回到调用它的地方。函数和过程类似，不同的是函数在返回到调用它的地方时要返回一个值。

如果你对C或C++很熟悉，会看到Pascal中的过程相当于C或C++中的函数返回void，而Pascal中的函数相当于C或C++函数返回一个值。

清单2-1演示了Pascal中的过程和函数。

清单2-1 过程和函数的例子

```
Program FuncProc;

{$APPTYPE CONSOLE}

procedure BiggerThanTen(i: Integer);
{ writes something to the screen if I is greater than 10 }
begin
  if I > 10 then
    writeln('Funky.');
end;

function IsPositive(I: Integer): Boolean;
{ Returns True if I is 0 or positive, False if I is negative }
begin
  if I < 0 then
    Result := False
  else
    Result := True;
end;

var
  Num: Integer;
begin
  Num := 23;
  BiggerThanTen(Num);
  if IsPositive(Num) then
    writeln(Num, 'Is positive.')
  else
    writeln(Num, 'Is negative.');
end.
```

注意 在IsPositive()函数中的本地变量Result需要特别注意。每一个Object Pascal函数都有一个隐含的本地变量称为Result，它包含了函数的返回值，注意这里和C和C++不一样，把一个值赋给Result，函数并不会结束。

你也能在函数体内把一个值赋给函数名来返回一个值，这是 Pascal的标准语法，是从Borland Pascal的老版本继承下来的。如果选择这种方法，一定要注意它和程序代码中把函数放在赋值运算符的左边是不同的！如果在函数体内把它放在赋值运算符的左边表示你要返回函数值，如果程序中把它放在左边表示要对它进行递归调用。

如果把Project|Options中的Compiler(编译)对话框中编译器的扩展语法(Extended Syntax)选项禁止了或在编译时用了{$X-}编译指令，则隐含变量Result不起作用。

传递参数

Pascal通过值或引用对函数和过程传递参数。传递的参数可以是基本类型、用户自定义类型或开放数组(开放数组将在本章介绍)。如果变量在过程和函数中不改变它的值也可以是常量。

1. 值参数

将参数以值的形式传递是默认的传递方式一个参数以值的形式传递意味着创建这个变量的本地副本，过程和函数对副本进行运算，看下面的例子：

```
procedure Foo(s: string);
```

当用这种方法调用一个过程时，一个字符串的副本就被创建， Foo()将对副本s进行运算，这表示对这个副本的任何修改都不会影响到原来的变量。

2. 引用参数

Pascal允许通过引用把变量传递给函数和过程。通过引用传递的参数有时又被称为变量参数，通过引用传递意味着接收变量的函数和过程能够改变变量的值。为了通过引用传递变量，在过程或函数的参数表中用关键字var:

```
procedure ChangeMe(var x: longint);
begin
  x := 3;  {x在调用过程中变了}
end;
```

不同于复制x，关键字var使得变量的地址被复制，因此变量值就能被直接改变。

用var的参数就像在C++中用&运算符通过引用传递变量一样。关键字 var把变量的地址传递给函数和过程，而不是把变量的值传递过去。

3. 常量参数

如果不想使传递给函数或过程的参数被改变，就用 const 关键字来声明它。关键字 const不仅保护了变量的值不被修改，而且对于传递给函数或过程的字符串和记录来说能产生更优化的代码，下面的代码就是一个过程声明接收一个字符串常量参数：

```
procedure Goon(const s: string);
```

4. 开放数组

开放数组参数能对过程和函数传递不确定数组，既可以传递相同类型的开放数组也可以传递不同类型的常量数组，下面的代码声明了一个函数，这个函数接受一个类型是 Integer的开放数组参数：

```
var
  i, Rez: Integer;
const
  j = 23;
begin
  i := 8;
  Rez := AddEmUp([i, 50, j, 89]);
```

为了在函数和过程中用开放数组，应用 High()、Low()和SizeOf() 等函数来获得关于开放数组的信息。下面的代码是AddEmUp()函数的实现，它返回传递给参数A的所有元素的总和：

```
function AddEmUp(A: array of Integer): Integer;
var
  i: Integer;
begin
  Result := 0;
  for i := Low(A) to High(A) do
    inc(Result, A[i]);
end;
```

Object Pascal 还支持常量数组(array of const)，这样能把不同类型的数据放在一个数组中传递给函数和过程。下面就是它的语法：

```
procedure WhatHaveIGot(A: array of const);
```

可以用下面的语法调用上述函数：

```
WhatHaveIGot(['Tabasco', 90, 5.6, @WhatHaveIGot, 3.14159, True, 's']);
```

实际上，传递时编译器把array of const参数的形式转换为TVarRec结构。TVarRec在System单元中是这样定义的：

```
type
PVarRec = ^TVarRec;
  TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
       vtInterface:  (VInterface: Pointer);
       vtWideString: (VWideString: Pointer);
       vtInt64:      (VInt64: PInt64);
  end;
```

其中VType域中的值指明了TVarRec所包容的数据类型，域中的值可能是下列值中的一个：

```
const
  { TVarRect.VType 的值 }
  vtInteger    = 0;
  vtBoolean    = 1;
  vtChar       = 2;
  vtExtended   = 3;
  vtString     = 4;
  vtPointer    = 5;
  vtPChar      = 6;
  vtObject     = 7;
  vtClass      = 8;
  vtWideChar   = 9;
  vtPWideChar  = 10;
  vtAnsiString = 11;
  vtCurrency   = 12;
  vtVariant    = 13;
  vtInterface  = 14;
  vtWideString = 15;
  vtInt64      = 16;
```

正如你所能想到的，由于array of const形式的参数可以传递不同类型的参数，使得接收这些参数的函数和过程工作起来比较困难。作为一个例子，下面的代码是 WhatHaveIGot()过程的实现，在这个例子中依次判断每一个数据的类型，并在屏幕上显示数据的序号和类型：

```
procedure WhatHaveIGot(A: array of const);
var
  i: Integer;
  TypeStr: string;
begin
  for i := Low(A) to High(A) do
  begin
```

```
      case A[i].VType of
        vtInteger    : TypeStr := 'Integer';
        vtBoolean    : TypeStr := 'Boolean';
        vtChar       : TypeStr := 'Char';
        vtExtended   : TypeStr := 'Extended';
        vtString     : TypeStr := 'String';
        vtPointer    : TypeStr := 'Pointer';
        vtPChar      : TypeStr := 'PChar';
        vtObject     : TypeStr := 'Object';
        vtClass      : TypeStr := 'Class';
        vtWideChar   : TypeStr := 'WideChar';
        vtPWideChar  : TypeStr := 'PWideChar';
        vtAnsiString : TypeStr := 'AnsiString';
        vtCurrency   : TypeStr := 'Currency';
        vtVariant    : TypeStr := 'Variant';
        vtInterface  : TypeStr := 'Interface';
        vtWideString : TypeStr := 'WideString';
        vtInt64      : TypeStr := 'Int64';
      end;
      ShowMessage(Format('Array item %d is a %s', [i, TypeStr]));
    end;
  end;
```

2.13 作用域

作用域是指一个过程、函数和变量能被编译器识别的范围，例如，一个全局常量的作用域是整个程序，而一些过程中的局部变量的作用域是那些过程。看清单2-2。

清单2-2 作用域的演示

```
program Foo;

{$APPTYPE CONSOLE}

const
  SomeConstant = 100;

var
  SomeGlobal: Integer;
  R: Real;

procedure SomeProc(var R: Real);
var
  LocalReal: Real;
begin
  LocalReal := 10.0;
  R := R - LocalReal;
end;

begin
  SomeGlobal := SomeConstant;
  R := 4.593;
  SomeProc(R);
end.
```

SomeConstant、SomeGlobal和R是全局变量，它们在程序的任何地方都能被编译器所识别。过程SomeProc()有两个变量R和LocalReal，它们的作用域是这个过程。如果试图在SomeProc()过程外访问

LocalReal，编译器将显示有未知识别符的错误。如果在SomeProc()中访问R，用的是局部变量的R，如果在这个过程外用R，则用的是全局变量的R。

2.14 单元

单元(unit)是组成Pascal程序的单独的源代码模块，单元由函数和过程组成，这些函数和过程能被主程序调用。一个单元至少要由以下三部分组成：

- 一个unit语句，每一个单元都必须在开头有这样一条语句，以标识单元的名称，单元的名称必须和文件名相匹配。例如，如果有一个文件名为FooBar，则unit语句可能是：

```
unit FooBar;
```

- interface部分，在unit语句后的源代码必须是interface语句。在这条语句和implementation语句之间是能被程序和其他单元所共享的信息。一个单元的interface部分是声明类型、常量、变量、过程和函数的地方，这些都能被主程序和其他单元调用。这里只能有声明，而不能有过程体和函数体。interface语句应当只有一个单词且在一行：

```
interface
```

- implementation部分，它在interface部分的后面。虽然单元的implementation包含了过程和函数的源代码，但它同时也允许在此声明不被其他单元所调用的任何数据类型、常量和变量。 implementation是定义在interface中声明的过程和函数的地方， implementation语句只有一个单词并且在一行上：

```
implementation
```

一个单元能可选地包含其他两个部分：

- initialization部分，在单元中它放在文件结尾前，它包含了用来初始化单元的代码，它在主程序运行前运行并只运行一次。
- finalization部分，在单元中它放在initialization和end之间。finalization部分是在Delphi 2.0引进的，在Delphi 1.0中这部分是用函数AddExitProc()增加一个退出过程来实现的，如果要把Delphi 1.0的程序移植过来，应该把退出过程中的代码移到这部分来。

注意 如果几个单元都有initalization/finalization部分，则它们的执行顺序与单元在主程序的users子句中的出现顺序一致。不要使intialization/finalization部分的代码依赖于它们的执行顺序，因为这样的话主程序的uses子句只要有小小的修改，就会导致程序无法通过编译。

2.14.1 uses子句

users子句在一个程序或单元中用来列出想要包含进来的单元。例如，如果有一个程序名为FooProg，它要用到在两个单元UnitA 和UnitB中的函数和类型，正确的uses声明应该这样：

```
Program FooProg;

uses UnitA, UnitB;
```

单元能有两个uses子句，一个在interface部分，一个在implementation部分。 这里有一个例子：

```
Unit FooBar;

interface

uses BarFoo;

  { 在这里进行全局声明 }
```

```
implementation

uses BarFly;

  { 在这里进行局部声明 }

initialization
  { 在这里进行单元的初始化 }
finalization
  { 在这里进行单元的退出操作 }
end.
```

2.14.2 循环单元引用

你经常会碰到这样的情况，在UnitA中调用UnitB并在UnitB中调用UnitA，这就被称为循环单元引用。循环单元引用的出现表明了程序设计有缺陷，应该在程序中避免使用循环单元引用。比较好的解决方法是把UnitA和UnitB共有的代码移到第三个单元中。然而，有时确实需要用到循环单元引用，这时就必须把一个uses子句移到implementation部分，而把另一个留在interface部分，这样就能解决问题。

2.15 包

Delphi的包能把应用程序的部分代码放到一个单独的模块中，它能被多个应用程序所共享。如果你有用Delphi 1.0和Delphi 2.0开发的代码，利用包技术，在不修改任何代码的情况下就能把它们利用起来。

可以把包看作是若干个单元集中在一起以类似于 DLL的形式存储的模块(Borland Package Library或BPL文件)。应用程序在运行的时候链接上这些包中的单元而不是在编译 /链接时，因为这些单元的代码存在于BPL文件中而不是存在于EXE或DLL中，所以EXE和DLL的长度将变得更短。

你能创建并使用下列四种类型的包：

- 运行期包，这种类型的包中的单元只有在应用程序运行时才被调用。当应用程序运行时，需要找到它所使用的运行期的包，否则无法正确执行。 Delphi的VCL50.DPL包就是典型的例子。
- 设计期包，这种类型的包包含了组件、属性和组件编辑器等在设计程序时需要的元素，可以用Component|Install Package命令将一个设计期的包安装到组件库中， Delphi的DCL*.BPL包就是这种类型的典型例子。这种类型的包在第21章“ 编写自定义组件 ”中有详细介绍。
- 既是运行期又是设计期的包，这种包结合了上述两种类型。创建这种类型的包使得应用程序的开发和分发变得简单，但这种类型的包的效率不是很高，因为它在分发版本中携带太多支持设计的内容。
- 既不是运行期又不是设计期的包，这种类型的包很少见，通常被其他包引用，而不是直接被应用程序引用。

2.15.1 使用Delphi的包

要让包技术支持应用程序很简单，只要在 Project|Options的Packages对话框中选中 Build with Runtime Packages复选框，以后当编译和运行应用程序时，应用程序就会动态地链接运行期包的单元，而不是把包的单元静态地链接到EXE和DLL中，这将使应用程序更加简洁(当然，分发程序时需要同时分发用到的运行期包)。

2.15.2 包的语法

包通常是用包编辑器创建的。要启动包编辑器，可以使用 File|New|Package菜单项。包编辑器产生

一个Delphi包源文件(DPK)，它将被编译进一个包。DPK的语法相当简单，其格式如下：

```
package PackageName

requires Package1, Package2, ...;

contains
  Unit1 in 'Unit1.pas',
  Unit2, in 'Unit2.pas',

  ...;
end.
```

列在requires子句中的包是这个包需要调用的其他包。通常在 contains子句下列出的单元是这个包所包含的单元，在contains子句下列出的单元将被编译进这个包，注意在这里列出的单元不能同时被列在requires子句中的包所包含。另外，由这些单元所引用的单元也会间接地包含到包中，除非它们已经被列在requires 子句中。

2.16 面向对象编程

关于面向对象编程(OOP)的文献已经有很多了，与其说 OOP是一种编程方法，还不如说它是一种信仰，由此引发了大量关于它的优点和缺点的讨论。我们不是正统的 OOP主义者，也不打算讨论OOP有什么优势；我们只是简单地告诉你面向对象编程的基本原则，这是 Object Pascal语言的基础。

OOP是使用独立对象(包含数据和代码)作为应用程序模块的范例。虽然 OOP不能使得代码容易编写，但它能使代码易于维护。将数据和代码结合在一起，能使定位和修复错误的工作得到简化，并最大限度地减少了对其他对象的影响，提高了程序的性能。通常，一门面向对象的编程语言至少要实现下列三个OOP的概念：

- 封装，把相关的数据和代码结合在一起，并隐藏了实现细节。封装的好处是有利于程序的模块化，并把代码和其他代码分开。
- 继承，是指一个新的对象能够从父对象中获取属性和方法，这种概念能用来建立 VCL这样的多层次的对象，首先建立通用对象，然后创建这些通用对象的有专用功能的子对象。

继承的好处是能共享代码。图2-4是一个继承的例子，根结点是Fruit，它是所有水果的祖先；其中包括Melons(瓜)，它是所有瓜的祖先；Melons中包括watermelon。

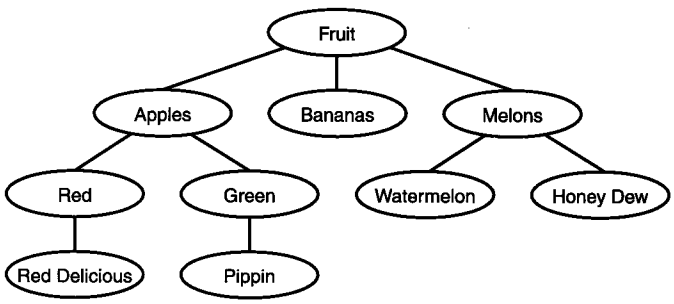

图2-4 继承的图示

- 多态性，从字面上说，是指多种形状。调用一个对象变量的方法时，实际被调用的代码与实际在变量中的对象的实例有关。

关于多继承

Object Pascal不像C++那样支持多继承，多继承是指一个对象能继承两个不同的对象，并

包含有两个父对象的所有数据和代码。

对图2-4继续扩展，用多继承来建立一个candy apple(苹果糖)对象，它继承于apple(苹果)类和其他一个称为candy(糖)的类。虽然这个功能看起来有用，但它包含着许多问题。

Object Pascal提供了两种方法来解决这个问题。第一种方法是在一个类中包含其他的类，你能从Delphi的VCL中看到这种解决方法。为了创建candy apple，先使candy对象称为apple对象的一个成员。第二种方法是用接口，在2.18.7节“接口”中将更详细地介绍接口，用了接口你能创建一个既支持candy接口又支持apple接口的对象。

在你继续深入了解对象的概念以前，应该先理解下面三个术语：

- 域(field)，也被称为域定义或实例变量，域是包含在对象中的数据变量。在对象中的一个域就像是在Pascal记录中一个域，在C++中它被称为数据成员。
- 方法(method)，属于一个对象的过程和函数名，在C++中它被称为成员函数。
- 属性(property)，属性是外部代码访问对象中的数据和代码的访问器，属性隐藏了一个对象的具体实现的细节。

注意 最好不要直接访问对象的域，因为实现对象的细节可能改变。相反用访问器属性来访问对象，它不受对象细节的影响，属性在本章的2.18.2节“属性”介绍。

基于对象与面向对象的编程

在某些工具中，你能操纵对象但不能创建对象， Visual Basic中的ActiveX 控件(以前称为OCX)就是例子。虽然在程序中你能用ActiveX控件，但你不能创建它，也不能派生它。这样的环境被称为基于对象的环境。

Delphi是完全的面向对象的环境，这表示在 Delphi中你能用已经存在的组件创建新的对象，这些对象是可视的或不可视的，甚至可以是设计时的窗体。

2.17 使用Delphi对象

前面提到，对象(也被称为类)是包括数据和代码的实体， Delphi的对象通过全面支持继承、封装和多态性，提供了面向对象编程的强大功能。

2.17.1 声明和实例化

在使用一个对象前，用class关键字声明一个对象。正如前面所提到的，在一个程序或单元的 type部分声明一个对象：

```
type
  TFooObject = class;
```

除了声明一个对象的类型，通常还需要一个对象的变量，即实例。实例定义在 var部分：

```
var
  FooObject: TFooObject;
```

在Object Pascal中通过调用它的一个构造器来建立一个对象的实例，构造器主要用来为对象创建实例并为对象中的域分配内存并进行初始化使得对象处在可以使用的状态。 Object Pascal的对象至少有一个构造器称为Create()，但一个对象可以有多个构造。根据不同的对象类型， Create()可以有不同的参数，本章主要集中在最简单的情况即不带参数的Create()。

不像在C++中，在Object Pascal中构造器不能自动调用，程序员必须自己调用构造器，调用构造器的语法如下：

```
FooObject := TFooObject.Create;
```

注意这里调用构造器的语法有点特殊，是通过类型来引用一个对象的 Create()方法，而不是像其他方法那样通过实例来引用。这看上去有点奇怪，但很有意义。变量 FooObject在调用是还没有定义，而TFooObject已经静态地存在于内存中，静态调用它的Create()方法是合法的。

通过调用构造器来创建对象的实例，这就是所谓的实例化。

注意 当一个对象实例用构造器创建时，编译器将对对象的每一个域进行初始化，你可以放心地认为所有数字被赋值为0，所有指针为nil，所有字符串为空。

2.17.2 析构

当用完了对象，应该调用这个实例的 Free()方法来释放它。Free()首先进行检查保证这个对象实例不为Nil，然后它调用对象的析构方法Destroy()。自然，析构进行与构造相反的工作，它释放所有分配的空间，并执行一些其他操作以保证对象能被适当地移出内存。语法是简单的：

```
FooObject.Free;
```

不像调用Create()，这里是调用对象实例的 Free()方法，记住不要直接调用 Destroy()，而调用更安全的Free()方法。

警告 在C++中，一个静态声明的对象在离开它的作用域时自动调用它的析构方法，但要对动态生成的对象手动调用析构方法。这个规则在Object Pascal中也是适用的，除了在Object Pascal中的隐式动态创建的对象，所以一定要记住这个规则：凡是创建的，都需要释放。对这个规则有两条重要的特例，第一条是当对象被其他对象拥有时(就像在第20章“VCL元素和运行期类型信息”中介绍的一样)，它将替你释放对象。第二种情况是引用计数的对象(像TInterfaceObject和TComObject)，当最后一个引用释放时，它将被析构。

你可能要问，这些方法是怎样进到对象中的，不需要为这些方法写代码，对吗？是的，刚才讨论的方法来自于Object Pascal的基类TObject对象。在Object Pascal中所有的对象都是TObject对象的后代，而不管它们是否是这样声明的。因此，下面的声明：

```
Type TFoo = Class;
```

相当于声明成：

```
Type TFoo = Class(TObject);
```

2.18 方法

方法是属于一个给定对象的过程和函数，方法反映的是对象的行为而不是数据，刚才我们提到了对象的两个重要的方法即构造器和析构方法。为了使对象能执行各种功能，你能在对象中定制方法。

创建一个方法用两个步骤，首先在对象类型的声明中声明这个方法。然后用代码定义方法。下面的代码就演示了声明和定义一个方法的步骤：

```
type
  TBoogieNights = class
    Dance: Boolean;
    procedure DoTheHustle;
  end;
procedure TBoogieNights.DoTheHustle;
begin
  Dance := True;
end;
```

注意，在定义方法体时，必须用完整的名字，就像在定义方法 DoTheHustle()时那样。同时也要注

意到，在这个方法中，对象的Dance域能被直接访问。

2.18.1 方法的类型

对象的方法能定义成静态(static)、虚拟(virtual)、动态(dynamic)或消息处理(message)。请看下面的例子：

```
TFoo = class
  procedure IAmAStatic;
  procedure IAmAVirtual; virtual;
  procedure IAmADynamic; dynamic;
  procedure IAmAMessage(var M: TMessage); message wm_SomeMessage;
end;
```

1. 静态方法

IAmAStatic是一个静态方法，静态方法是方法的缺省类型，对它就像对通常的过程和函数那样调用。编译器知道这些方法的地址，所以调用一个静态方法时它能把运行信息静态地链接进可执行文件。静态方法执行的速度最快，但它们却不能被覆盖来支持多态性。

2. 虚拟方法

IAmAVirtual是一个虚拟方法。虚拟方法和静态方法的调用方式相同。由于虚拟方法能被覆盖，在代码中调用一个指定的虚拟方法时编译器并不知道它的地址。因此，编译器通过建立虚拟方法表(VMT)来查找在运行时的函数地址。所有的虚拟方法在运行时通过 VMT来调度，一个对象的VMT表中除了自己定义的虚拟方法外，还有它的祖先的所有的虚拟方法，因此虚拟方法比动态方法用的内存要多，但它执行得比较快。

3. 动态方法

IAmADynamic是一个动态方法，动态方法跟虚拟方法基本相似，只是它们的调度系统不同。编译器为每一个动态方法指定一个独一无二的数字，用这个数字和动态方法的地址构造一个动态方法表(DMT)。不像VMT表，在DMT表中仅有它声明的动态方法，并且这个方法需要祖先的 DMT表来访问它其余的动态方法。正因为这样，动态方法比虚拟方法用的内存要少，但执行起来较慢，因为有可能要到祖先对象的DMT中查找动态方法。

4. 消息处理方法

IAmAMessage是一个消息处理方法，在关键字 message后面的值指明了这个方法要响应的消息。用消息处理方法来响应Windows的消息，这样就不用直接来调用它。消息处理的详细内容在第 5章“理解Windows消息”讨论。

5. 方法的覆盖

在Object Pascal覆盖一个方法用来实现OOP的多态性概念。通过覆盖使一方法在不同的派生类间表现出不同的行为。Object Pascal中能被覆盖的方法是在声明时被标识为 virtual或dynamic的方法。为了覆盖一个方法，在派生类的声明中用 override代替virtual或dynamic。例如，能用下面的代码覆盖IAmAVirtual和IAmADynamic方法：

```
TFooChild = class(TFoo)
  procedure IAmAVirtual; override;
  procedure IAmADynamic; override;
  procedure IAmAMessage(var M: TMessage); message wm_SomeMessage;
end;
```

用了override关键字后，编译器就会用新的方法替换VMT中原先的方法。如果用virtaul或dynamic替换override重新声明IAmAVirtual和IAmADynamic，将是建立新的方法而不是对祖先的方法进行覆盖。同样，在派生类中如果企图对一个静态方法进行覆盖，在新对象中的方法完全替换在祖先类中的同名方法。

6. 方法的重载

就像普通的过程和函数，方法也支持重载，使得一个类中有许多同名的方法带着不同的参数表，能重载的方法必须用overload指示符标识出来，可以不对第一个方法用 overload。下面的代码演示了一个类中有三个重载的方法：

```
type
  TSomeClass = class
    procedure AMethod(I: Integer); overload;
    procedure AMethod(S: string); overload;
    procedure AMethod(D: Double); overload;
  end;
```

7. 重新引入方法名称

有时候，需要在派生类中增加一个方法，而这个方法的名称与祖先类中的某个方法名称相同。在这种情况下，没必要覆盖这个方法，只要在派生类中重新声明这个方法。但在编译时，编译器就会发出一个警告，告诉你派生类的方法将隐藏祖先类的同名方法。要解决这个问题，可以在派生类中使用reintroduce指示符，下面的代码演示了reintroduce指示符的正确用法：

```
type
  TSomeBase = class
    procedure Cooper;
  end;

  TSomeClass = class
    procedure Cooper; reintroduce;
  end;
```

8. Self

在所有对象的方法中都有一个隐含变量称为 Self，Self是用来调用方法的指向类实例的指针。 Self由编译器作为一个隐含参数传递给方法。

2.18.2 属性

可以把属性看成是能对类中的数据进行修改和执行代码的特殊的辅助域。对于组件来说，属性就是列在Object Inspector窗口的内容。下面的例子定义了一个有属性的简单对象：

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
public
  property Value: Integer read SomeValue write SetSomeValue;
end;
procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
    SomeValue := AValue;
end;
```

TMyObject是包含下列内容的对象：一个域（被称为SomeValue的整型数）、一个方法（被称为SetSomeValue的过程）和一个被称为value的属性。SetSomeValue过程的功能是对 SomeValue域赋值，Value属性实际上不包含任何数据。 Value是SomeValue域的辅助域，当想得到Value中的值时，它就从SomeValue读值，当试图对Value属性设置值时，Value就调用SetSomeValue对SomeValue设置值。这样做的好处有两个方面：首先，通过一个简单变量就使得外部代码可以访问对象的数据，而不需要知道对象的实现细节。其次，在派生类中可以覆盖诸如 SetSomeValue的方法以实现多态性。

2.18.3 可见性表示符

Object Pascal能通过在声明域和方法时用portected、private，public，published和automated指示符来对对象提供进一步的控制。使用这些关键字的语法如下：

```
TSomeObject = class
private
  APrivateVariable: Integer;
  AnotherPrivateVariable: Boolean;
protected
  procedure AProtectedProcedure;
  function ProtectMe: Byte;
public
  constructor APublicContructor;
  destructor APublicKiller;
published
  property AProperty read APrivateVariable write APrivateVariable;
end;
```

在每一个指示符下能声明任意多个方法或域。书写时要注意缩进格式。下面是这些指示符的含义：

- private，对象中的这部分只能被相同单元的代码访问。用这个指示符对用户隐藏了对象实现的细节并阻止用户直接修改对象中的敏感部分。
- protected，对象中的这部分成员能被它的派生类访问，这样不仅能使对象向用户隐藏实现的细节并为对象的派生类提供了最大的灵活性。
- public，这部分的域和方法能在程序的任何地方访问，对象的构造器和析构方法通常应该是public。
- published，对象的这一部分将产生运行期类型信息 (RTTI)，并使程序的其他部分能访问这部分。Object Inspector用RTTI来产生属性的列表。
- automated，这个指示符其实已经不用了，保留这个指示符的目的是为了与Delphi 2.0的代码兼容，第23章详细地介绍其细节。

下面的代码是以前介绍过的TMyObject对象，其中通过增加指示符提高了对象的完整性：

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
published
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
    SomeValue := AValue;
end;
```

现在，对象的用户不能直接修改 SomeValue的值了，要修改对象的数据就必须通过 Value属性来实现。

2.18.4 友类

在C++语言中有友类的概念(允许在其他类中访问私有数据和私有函数的类)。在C++中这是通过关键字friend来实现的，严格地说，在Object Pascal中没有类似的关键字，但有类似的功能。凡是在相同

单元声明的对象都认为是友类，都可以访问其他对象的私有成员。

2.18.5 对象的秘密

在Object Pascal中的类实例实际上是指向堆中的类实例数据的32位指针。当访问对象的域、方法和属性时，编译器会自动产生一些代码来处理这个指针。因此对于新手来说，对象就好像是一个静态变量。这意味着，Object Pascal无法像C++那样在应用程序的数据段中为类分配内存，而只能在堆中分配内存。

2.18.6 TObject：所有对象的祖先

因为所有对象都是从TObject继承来的，每一个类都从TObejct继承了一些方法，所以可以对对象的性能进行一些特殊的假定。每一个类都能告诉你它的名字、类型和它是否从某个类派生而来。作为一个程序员，不必关心编译器的实现细节而只要能利用对象所提供的功能就够了。

TObject是一个特殊的对象，它在 system单元中定义，编译器对 TObject是完全清楚的，下面是TObject的定义：

```
type
  TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString): Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID; out Obj): Boolean;
    class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  end;
```

在Delphi的联机帮助中你将看到每一个方法的文档。

在这里特别要注意那些前面有 class关键字的方法。在一个方法前加上关键字 class，使得方法向其他通常的过程和函数一样调用而不需要生成一个包含这个方法的类的实例，这个功能是从 C++的static函数借鉴来的。要小心，不要让一个类方法依赖于任何实例信息，否则编译时将出错。

2.18.7 接口

对于Object Pascal语言来说，最近一段时间最有意义的改进就是从Delphi 3开始支持接口(interface),

接口定义了能够与一个对象进行交互操作的一组过程和函数。对一个接口进行定义包含两个方面的内容，一方面是实现这个接口，另一方面是定义接口的客户。一个类能实现多个接口，即提供多个让客户用来控制对象的“表现方式”。

正如名字所表现的，一个接口就是对象和客户通信的接口。这个概念像 C++中的PURE VIRTUAL 类。实现接口的函数和过程是支持这个接口的类的工作。

在本章你将学到接口的语言元素，要想在应用程序中使用接口，请参考第 23章“COM和ActiveX”；

1. 定义接口

就像所有的Delphi类都派生于TObject一样，所有的接口都派生于一个被称为是 IUnknown的接口，IUnknown在system单元中定义如下：

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

正如你所看到的，接口的定义就像是类的定义，最根本的不同是在接口中有一个全局唯一标识符(GUID)，它对于每一个接口来说是不同的。对 IUnknown的定义来自于 Microsoft的组件对象模型(COM)规范。在第23章有详细的介绍。

如果你知道怎样创建 Delphi的类，那么定义一个定制的接口是一件简单的事情，下面的代码定义了一个新的接口称为IFoo，它包含一个被称为F1()的方法：

```
type
  IFoo = interface
    ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
    function F1: Integer;
  end;
```

提示 在Delphi的IDE中，按Ctrl+Shift+G键可以为一个接口生成一个新的GUID。

下面的代码声明了一个称为IBar的接口，它是从IFoo接口继承来的：

```
type
  IBar = interface(IFoo)
    ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
    function F2: Integer;
  end;
```

2. 实现接口

下面的代码演示了在一个类TFooBar中怎样实现IFoo和IBar接口：

```
type
  TFooBar = class(TInterfacedObject, IFoo, IBar)
    function F1: Integer;
    function F2: Integer;
  end;

function TFooBar.F1: Integer;
begin
  Result := 0;
end;

function TFooBar.F2: Integer;
begin
```

```
  Result := 0;
end;
```

注意，一个类可以实现多个接口，只要在声明这个类时依次列出要实现的接口。编译器通过名称来把接口中的方法与实现接口的类中的方法对应起来，如果一个类只是声明要实现某个接口，但并没有具体实现这个接口的方法，编译将出错。

如果一个类要实现多个接口，而这些接口中包含同名的方法，必须把同名的方法另取一个别名，请看下面的程序示例：

```
type
  IFoo = interface
    ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
    function F1: Integer;
  end;

  IBar = interface
    ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
    function F1: Integer;
  end;

  TFooBar = class(TInterfacedObject, IFoo, IBar)
    // 为同名方法取别名
    function IFoo.F1 = FooF1;
    function IBar.F1 = BarF1;
    // 接口方法
    function FooF1: Integer;
    function BarF1: Integer;
  end;

function TFooBar.FooF1: Integer;
begin
  Result := 0;
end;

function TFooBar.BarF1: Integer;
begin
  Result := 0;
end;
```

3. implements指示符

implements指示符是在Delphi 4中引进的，它的作用是委托另一个类或接口来实现接口的某个方法，这个技术有时又被称为委托实现，关于implements指示符的用法，请看下面的代码：

```
type
  TSomeClass = class(TInterfacedObject, IFoo)
    // stuff
    function GetFoo: TFoo;
    property Foo: TFoo read GetFoo implements IFoo;
    // stuff
  end;
```

在上面例子中的implements指示符是要求编译器在Foo属性中寻找实现IFoo接口方法。属性的类型必须是一个类，它包含IFoo方法或类型是IFoo的接口或IFoo派生接口。implements指示符后面可以列出几个接口，彼此用逗号隔开。

implements指示符在开发中提供了两个好处：首先，它允许以无冲突的方式进行接口聚合。聚合(Aggregation)是COM中的概念。它的作用是把多个类合在一起共同完成一个任务，详见第23章。其次，它能够延后占用实现接口所需的资源，直到确实需要资源。例如，假设实现一个接口需要分配一个

1MB的位图，但这个接口很少用到。因此，可能平时你不想实现这个接口，因为它太耗费资源了，用implements指示符后，可以只在属性被访问时才创建一个类来实现接口。

4. 使用接口

当在应用程序中使用接口类型的变量时，要用到一些重要的语法规则。最需要记住的是，一个接口是生存期自管理类型的，这意味着，它通常被初始化为 nil，它是引用计数的，当获得一个接口时自动增加一个引用计数；当它离开作用域或赋值为 nil时它被自动释放。下面的代码演示了一个接口变量的生存期自管理机制。

```
var
  I: ISomeInterface;
begin
  // I被初始化为nil
  I := FunctionReturningAnInterface;  // I的引用计数加1
  I.SomeFunc;
  // I的引用计数减1，如果为0，则自动释放。
end;
```

关于接口变量的另一个规则是，一个接口变量与实现这个接口的类是赋值相容的，例如，下面的代码是合法的：

```
procedure Test(FB: TFooBar)
var

  F: IFoo;
begin

  F := FB;  // 合法，因为FB支持IFoo
  .
  .
  .
```

最后，类型强制转换运算符as可以把一个接口类型的变量强制类型转换为另一种接口 (在第23章详细介绍)。示例如下：

```
var
  FB: TFooBar;
  F: IFoo;
  B: IBar;
begin
  FB := TFooBar.Create
  F := FB;  // 合法，因为FB支持IFoo
  B := F as IBar;  // 把F转换为IBar
  .
  .
  .
```

2.19 结构化异常处理

结构化异常处理(SEH)是一种处理错误的手段，使得应用程序能够从致命的错误中很好地恢复。在Delphi 1.0中，异常是由Object Pascal语言来处理的；从Delphi 2.0开始，异常成为Win32 API的一部分。用Object Pascal来处理异常比较简单了，因为异常中包含了错误的位置和特征信息。这使得异常的使用和实现与普通的类一样。

Delphi中包含了一些预定义的通用的程序错误异常，例如内存不足、被零除、数字上溢和下溢以及文件的输入输出错误，你可以定义自己的异常类来适应程序的需要。清单 2-3演示了在文件输入/输出时，怎样用异常处理。

清单2-3 使用异常处理的文件输入/输出

```
Program FileIO;

uses Classes, Dialogs;

{$APPTYPE CONSOLE}

var
  F: TextFile;
  S: string;
begin
  AssignFile(F, 'FOO.TXT');
  try
    Reset(F);
    try
      ReadLn(F, S);
    finally
      CloseFile(F);
    end;
  except
    on EInOutError do
      ShowMessage('Error Accessing File!');
  end;
end.
```

在清单2-3内层的try...finally代码块用来确保文件总是关闭的，而不管是不是发生了异常。这段代码实际上就是“ Hey，程序，请执行try与finally之间的代码；如果执行完毕或出现异常，就执行 finally与end之间的代码；如果确实有异常发生，就跳到外层的异常处理块 ”。这样，即使出现异常，文件也总是关闭的，并且异常总能得到处理。

注意 在try...finally块中，finally后面的语句不管有没有异常都被执行。因此，finally后面的语句不能以发生异常为前提。另外，由于finally后面的语句并没有处理异常，因此，异常被传递到下一层的异常处理块。

外层的try...except块用于处理程序中发生的异常。在 finally中关闭文件后， except块显示一个信息，告诉用户发生了I/O错误。

这种异常处理机制比传统的错误处理方式优越，它使得错误检测代码从错误纠正代码中分离出来。这是一件好事情，它会使程序更可读，它使得你能集中处理程序的其他代码部分。

使用try...finally代码块但不捕捉特定种类的异常是有一定意义的。当代码中使用 try...finally块的时候，意味着程序并不关心是否发生异常，而只是想最终总是能进行某项任务。 finally块最适合于释放先前分配的资源(例如文件或Windows资源)，因为它总是执行的，即使发生了错误。不过，很多情况下，可能需要对特定的异常做特定的处理，这时候就要用 try...except块来捕捉特定的异常，清单 2-4就是例子。

清单2-4 一个try...except异常处理块

```
Program HandleIt;

{$APPTYPE CONSOLE}

var
  R1, R2: Double;
begin
```

```
  while True do begin
  try
    Write('Enter a real number: ');
    ReadLn(R1);
    Write('Enter another real number: ');
    ReadLn(R2);
    Writeln('I will now divide the first number by the second...');
    Writeln('The answer is: ', (R1 / R2):5:2);
  except
    On EZeroDivide do
      Writeln('You cannot divide by zero!');
    On EInOutError do
      Writeln('That is not a valid number!');
  end;
  end;
end.
```

尽管在try...except块中可以捕捉特定的异常，也可以用try...except...else结构来捕捉其他异常，请看下面的代码：

```
try
  Statements
except
  On ESomeException do Something;
else
  { 进行一些默认的异常处理 }
end;
```

注意 当使用try...except...else结构的时候，应当明白else部分会捕捉所有的异常，包括那些你并没有预料到的异常，例如内存不足或其他运行期异常。因此，使用else 部分要小心，能不用则不用。当进入不合格的异常处理过程中时，你应当一直重触发这个异常，这在 2.19.3节“ 重新触发异常 ”中介绍。

其实，下面的代码也能够达到类似于 try...except...else结构的效果，因为在except部分没有指定异常类。

```
try
  Statements
except
  HandleException   // 效果与try...except...else结构几乎相同
end;
```

2.19.1 异常类

异常是一种特殊的对象实例，它在异常发生时才实例化，在异常被处理后自动删除。异常对象的基类被称为Exception，它是这样声明的：

```
type
  Exception = class(TObject)
  private
    FMessage: string;
    FHelpContext: Integer;
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string; const Args: array of const);
    constructor CreateRes(Ident: Integer); overload;
    constructor CreateRes(ResStringRec: PResStringRec); overload;
    constructor CreateResFmt(Ident: Integer; const Args: array of const);
```

```
overload;
    constructor CreateResFmt(ResStringRec: PResStringRec; const Args: array of
const); overload;
    constructor CreateHelp(const Msg: string; AHelpContext: Integer);
    constructor CreateFmtHelp(const Msg: string; const Args: array of const;
      AHelpContext: Integer);
    constructor CreateResHelp(Ident: Integer; AHelpContext: Integer); overload;
    constructor CreateResHelp(ResStringRec: PResStringRec; AHelpContext:
Integer); overload;
    constructor CreateResFmtHelp(ResStringRec: PResStringRec; const Args: array
of const;
      AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(Ident: Integer; const Args: array of const;
      AHelpContext: Integer); overload;
    property HelpContext: Integer read FHelpContext write FHelpContext;
    property Message: string read FMessage write FMessage;
  end;
```

在异常对象中最重要的元素是 Message属性，它是一个字符串，它提供了对异常的解释，由Message所提供的信息是根据产生的异常来决定的。

注意 如果定义自己的异常对象，一定是要从一个已知的异常对象例如Exception或它的派生类派生出来的，因为这样通用的异常处理过程才能捕捉这个异常。

当你在except块中处理一个特定的异常时，可能会捕捉到该异常的派生异常。例如， EMathError是所有与数学有关的异常(例如EZeroDivide、EOverflow)的祖先，示例如下：

```
try
  Statements
except
  on EMathError do   // 将捕捉EMathError及其派生异常
    HandleException
end;
```

凡是没有显式地处理的异常最终将被传送到 Delphi运行期库中的默认处理过程并在此得到处理。这个默认的处理过程将打开一个消息框，告诉用户一个异常发生了。顺便说一下，第 4章“应用程序框架和设计”将给出程序示例来演示怎样覆盖默认的异常处理处理过程。

处理异常的时候，可能需要访问异常对象的实例，以便获得更多的有关异常的信息。要访问异常对象的实例有两种方法：一是在on ESomeException结构中使用可选的标识符，二是使用ExceptObject()函数。

可以在except块的on ESomeException部分插入一个可选的标识符并且具有一个到当前产生的异常的实例的标识符映射。这么做的语法是在异常类型前放上一个标识符和一个冒号，如下：

```
try
  Something
except
  on E:ESomeException do
    ShowMessage(E.Message);
end;
```

在这里，标识符 E成为当前产生的异常的实例。这个标识符的类型总是和它后面的异常类型一致。

也可以使用Exception()函数返回当前异常对象的实例，不过， ExceptObject()函数的返回类型是TObject，必须把它强制转换为需要的异常对象。下面的代码演示了ExceptObject()函数的用法：

```
try
  Something
```

```
except
  on ESomeException do
    ShowMessage(ESomeException(ExceptObject).Message);
end;
```

如果当前没有异常，ExceptObject()函数将返回nil。

触发一个异常的语法类似于创建一个对象实例。例如，要触发一个称为 EBadStuff的用户自定义异常，可以用下面的语法：

```
Raise EBadStuff.Create('Some bad stuff happened.');
```

2.19.2 执行的流程

在一个异常触发后，应用程序的流程就跳到了下一个异常处理过程，直到这个异常实例被处理结束并释放空间。执行的流程取决于调用栈，因此，跳转的范围是整个程序 (而不限于一个过程或单元)。清单2-5演示了当发生异常时程序的执行流程。从这个清单中可以看出，单元的名称为 main，它有一个窗体，窗体上有一个按钮。当单击按钮时， Button1Click()就调用Proc1()，而Proc1()又会调用Proc2()，Proc2()又会调用Proc3()。如果在Proc3()中发生了异常，你将看到程序的执行流程是怎样在 try...finally 块中跳转的，直到Button1Click()最终处理了异常。

提示 在Delphi的IDE中运行此程序时，最好不要选中Tools|Debugger Options|的对话框中的Stop on Delphi Exception 复选框。这样能更好地看到执行流程的变化情况。

清单2-5 异常传递项目的main单元

```
unit Main;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

type
  EBadStuff = class(Exception);
procedure Proc3;
begin
  try
```

```
    raise EBadStuff.Create('Up the stack we go!');
  finally
    ShowMessage('Exception raised. Proc3 sees the exception');
  end;
end;

procedure Proc2;
begin
  try
    Proc3;
  finally
    ShowMessage('Proc2 sees the exception');
  end;
end;

procedure Proc1;
begin
  try
    Proc2;
  finally
    ShowMessage('Proc1 sees the exception');
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
const
  ExceptMsg = 'Exception handled in calling procedure. The message is "%s"';
begin
  ShowMessage('This method calls Proc1 which calls Proc2 which calls Proc3');
  try
    Proc1;
  except
    on E:EBadStuff do
     ShowMessage(Format(ExceptMsg, [E.Message]));
  end;
end;

end.
```

2.19.3 重新触发异常

要对try...except中的一条语句进行特殊处理，并要使异常能够传递给外层的默认的处理过程时，需要重新触发异常处理。清单2-6演示了怎样重新触发异常。

清单2-6 再次触发异常

```
try              // this is outer block
  { statements }
  { statements }
  ( statements }
  try            // this is the special inner block
    { some statement that may require special handling }
  except
    on ESomeException do
    begin
      { special handling for the inner block statement }
      raise;     // reraise the exception to the outer block
```

```
    end;
  end;
except
  // outer block will always perform default handling
  on ESomeException do Something;
end;
```

2.20 运行期类型信息

运行期类型信息(RTTI)是一种语言特征，能使应用程序在运行时得到关于对象的信息。 RTTI是Delphi的组件能够融合到IDE中的关键。它在IDE中不仅仅是一个纯学术的过程。

由于对象都是从TObject继承下来的，因此，对象都包含一个指向它们的 RTTI的指针以及几个内建的方法。下面的表列出了TObject的一些方法，用这些方法能获得某个对象实例的信息。

函 数	返回类型	返 回 值
ClassName()	string	对象的类名
ClassType()	TClass	对象的类型
InheritsFrom()	Boolean	判断对象是否继承于一个指定的类
ClassParent()	TClass	对象的祖先类型
InstanceSize()	word	对象实例的长度(字节数)
ClassInfo()	Pointer	指向RTTI的指针

Object Pascal提供了两个运算符as和is，用它们通过RTTI能对对象进行比较和强制类型转换。

关键字as 是类型转换的一种新的形式。它能把一个基层的对象强制类型转换成它的派生类，如果转换不合法就产生一个异常。假定有一个过程，想让它能够传递任何类型的对象，它应该这样定义：

```
Procedure Foo(AnObject: TObject);
```

在这个过程如果要对AnObject进行操作，要把它转换为一个派生对象。假定把 AnObject看成是一个TEdit派生类型，并想要改变它所包含的文本(TEdit是一个Delphi VCL编辑控件)，用下列代码：

```
(Foo as TEdit).Text := 'Hello World.';
```

能用比较运算符来判断两个对象是否是相兼容的类型，用 is 运算符把一个未知的对象和一个已知类型或实例进行比较，确定这个未知对象的属性和行为。例如，在对 AnObject进行强制类型转换前，确定AnObject和TEdit是否指针兼容：

```
If (Foo is TEdit) then
  TEdit(Foo).Text := 'Hello World.';
```

注意在这个例子中不能用as进行强制类型转换，这是因为它要大量使用 RTTI，另外还因为，在第一行已经判断Foo就是TEdit，可以通过在第2行进行指针转换来优化。

2.21 总结

本章的内容相当丰富，你学习了Object Pascal语言的基本语法和语义，其中包括：变量、运算符、函数、过程和类型，同时对面向对象编程、对象、域、属性、方法、 TObject、接口、异常处理和RTTI有了了解。

现在，你已经明白了面向对象的Object Pascal是怎样工作的，可以进行更深入地讨论Win32API和可视化组件库(VCL)。