The Object Pascal Language

CHAPTER

2

IN THIS CHAPTER

This chapter sets aside the visual elements of Delphi in order to provide you with an overview of Delphi's underlying language—Object Pascal. To begin with, you'll receive an introduction to the basics of the Object Pascal language, such as language rules and constructs. Later on, you'll learn about some of the more advanced aspects of Object Pascal, such as classes and exception handling. Because this isn't a beginner's book, it assumes that you have some experience with other high-level computer languages such as C, C++, or Visual Basic, and it compares Object Pascal language structure to that of those other languages. By the time you're finished with this chapter, you'll understand how programming concepts such as variables, types, operators, loops, cases, exceptions, and objects work in Pascal as compared to C++ and Visual Basic.

Even if you have some recent experience with Pascal, you'll find this chapter useful, as this is really the only point in the book where you learn the nitty-gritty of Pascal syntax and semantics.

Comments

As a starting point, you should know how to make comments in your Pascal code. Object Pascal supports three types of comments: curly brace comments, parentheses/asterisk comments, and C++-style double backslash comments. Examples of each type of comment follow:

```
{ Comment using curly braces }

(* Comment using paren and asterisk *)

// C++-style comment
```

The two types of Pascal comments are virtually identical in behavior. The compiler considers the comment to be everything between the open-comment and close-comment delimiters. For C++-style comments, everything following the double backslash until the end of the line is considered a comment.

NOTE

You cannot nest comments of the same type. Although it is legal syntax to nest Pascal comments of different types inside one another, we don't recommend the practice. Here are some examples:

```
{ (* This is legal *) }
(* { This is legal } *)
(* (* This is illegal *) *)
{ { This is illegal } }
```

New Procedure and Function Features

Because procedures and functions are fairly universal topics as far as programming languages are concerned, we won't go into too much detail here. We just want to fill you in on a few new or little-known features.

Parentheses

Although not new to Delphi 5, one of the lesser-known features of Object Pascal is that parentheses are optional when calling a procedure or function that takes no parameters. Therefore, the following syntax examples are both valid:

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

Granted, this feature isn't one of those things that sends chills up and down your spine, but it's particularly nice for those who split their time between Delphi and languages such as C++ or Java, where parentheses are required. If you're not able to spend 100 percent of your time in Delphi, this feature means you don't have to remember to use different function-calling syntax for different languages.

Overloading

Delphi 4 introduced the concept of function overloading (that is, the ability to have multiple procedures or functions of the same name with different parameter lists). All overloaded methods are required to be declared with the overload directive, as shown here:

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

Note that the rules for overloading methods of a class are slightly different and are explained in the section "Method Overloading." Although this is one of the features most requested by developers since Delphi 1, the phrase that comes to mind is, "Be careful what you wish for." Having multiple functions and procedures with the same name (on top of the traditional ability to have functions and procedures of the same name in different units) can make it more difficult to predict the flow of control and debug your application. Because of this, overloading is a feature you should employ judiciously. Not to say that you should avoid it; just don't overuse it.

Default Value Parameters

Also introduced in Delphi 4 were default value parameters (that is, the ability to provide a default value for a function or procedure parameter and not have to pass that parameter when calling the routine). In order to declare a procedure or function that contains default value parameters, follow the parameter type with an equal sign and the default value, as shown in the following example:

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

The `HasDefVal()` procedure can be called in one of two ways. First, you can specify both parameters:

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

Second, you can specify only parameter `S` and use the default value for `I`:

```
HasDefVal('hello');  // default value used for I
```

You must follow several rules when using default value parameters:

- Parameters having default values must appear at the end of the parameter list. Parameters without default values may not follow parameters with default values in a procedure or function's parameter list.
- Default value parameters must be of an ordinal, pointer, or set type.
- Default value parameters must be passed by value or as `const`. They may not be reference (`out`) or untyped parameters.

One of the biggest benefits of default value parameters is in adding functionality to existing functions and procedures without sacrificing backward compatibility. For example, suppose you sell a unit that contains a revolutionary function called `AddInts()` that adds two numbers:

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

In order to keep up with the competition, you feel you must update this function so that it has the capability for adding three numbers. However, you're loathe to do so because adding a parameter will cause existing code that calls this function to not compile. Thanks to default parameters, you can enhance the functionality of `AddInts()` without compromising compatibility. Here's an example:

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

Variables

You might be used to declaring variables off the cuff: "I need another integer, so I'll just declare one right here in the middle of this block of code." If that has been your practice, you're going to have to retrain yourself a little in order to use variables in Object Pascal. Object Pascal requires you to declare all variables up front in their own section before you begin a procedure, function, or program. Perhaps you used to write free-wheeling code like this:

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

In Object Pascal, any such code must be tidied up and structured a bit more to look like this:

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

NOTE

Object Pascal—like Visual Basic, but unlike C and C++—is not a case-sensitive language. Upper- and lowercase is used for clarity's sake, so use your best judgment, as the style used in this book indicates. If the identifier name is several words mashed together, remember to capitalize for clarity. For example, the following name is unclear and difficult to read:

```
procedure thisprocedurenamemakesnosense;
```

This code is quite readable, however:

```
procedure ThisProcedureNameIsMoreClear;
```

For a complete reference on the coding style guidelines used for this book, see Chapter 6, "Coding Standards," on the CD accompanying this book.

You might be wondering what all this structure business is and why it's beneficial. You'll find, however, that Object Pascal's structured style lends itself to code that's more readable, maintainable, and less buggy than the more scattered style of C++ or Visual Basic.

Notice how Object Pascal enables you to group more than one variable of the same type together on the same line with the following syntax:

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

Remember that when you're declaring a variable in Object Pascal, the variable name precedes the type, and there's a colon between the variables and types. Note that the variable initialization is always separate from the variable declaration.

A language feature introduced in Delphi 2 enables you to initialize global variables inside a `var` block. Here are some examples demonstrating the syntax for doing so:

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

NOTE

Preinitialization of variables is only allowed for global variables, not variables that are local to a procedure or function.

TIP

The Delphi compiler sees to it that all global data is automatically zero-initialized. When your application starts, all integer types will hold `0`, floating-point types will hold `0.0`, pointers will be `nil`, strings will be empty, and so forth. Therefore, it's not necessary to zero-initialize global data in your source code.

Constants

Constants in Pascal are defined in a `const` clause, which behaves similarly to C's `const` keyword. Here's an example of three constant declarations in C:

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

The major difference between C constants and Object Pascal constants is that Object Pascal, like Visual Basic, does not require you to declare the constant's type along with the value in the declaration. The Delphi compiler automatically allocates proper space for the constant based on its value, or, in the case of scalar constants such as Integer, the compiler keeps track of the values as it works, and space never is allocated. Here's an example:

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

NOTE

Space is allocated for constants as follows: Integer values are "fit" into the smallest type allowable (10 into a ShortInt, 32,000 into a SmallInt, and so on). Alphanumeric values fit into Char or the currently defined (by $H) string type. Floating-point values are mapped to the extended data type, unless the value contains four or fewer decimal places explicitly, in which case it's mapped to a Comp type. Sets of Integer and Char are of course stored as themselves.

Optionally, you can also specify a constant's type in the declaration. This provides you with full control over how the compiler treats your constants:

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

Object Pascal permits the usage of compile-time functions in const and var declarations. These routines include Ord(), Chr(), Trunc(), Round(), High(), Low(), and SizeOf(). For example, all of the following code is valid:

```
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);
```

CAUTION

The behavior of 32-bit Delphi type-specified constants is different from that in 16-bit Delphi 1. In Delphi 1, the identifier declared wasn't treated as a constant but as a preinitialized variable called a *typed constant*. However, in Delphi 2 and later, type-specified constants have the capability of being truly constant. Delphi provides a backward-compatibility switch on the Compiler page of the Project, Options dialog box, or you can use the `$J` compiler directive. By default, this switch is enabled for compatibility with Delphi 1 code, but you're best served not to rely on this capability because the implementers of the Object Pascal language are trying to move away from the notion of assignable constants.

If you try to change the value of any of these constants, the Delphi compiler emits an error explaining that it's against the rules to change the value of a constant. Because constants are read-only, Object Pascal optimizes your data space by storing those constants that merit storage in the application's code pages. If you're unclear about the notions of code and data pages, see Chapter 3, "The Win32 API."

NOTE

Object Pascal does not have a preprocessor as do C and C++. There's no concept of a macro in Object Pascal and, therefore, no Object Pascal equivalent for C's `#define` for constant declaration. Although you may use Object Pascal's `$define` compiler directive for conditional compiles similar to C's `#define`, you cannot use it to define constants. Use `const` in Object Pascal where you would use `#define` to declare a constant in C or C++.

Operators

Operators are the symbols in your code that enable you to manipulate all types of data. For example, there are operators for adding, subtracting, multiplying, and dividing numeric data. There are also operators for addressing a particular element of an array. This section explains some of the Pascal operators and describes some of the differences between their C and Visual Basic counterparts.

Assignment Operators

If you're new to Pascal, Delphi's assignment operator is going to be one of the toughest things to get used to. To assign a value to a variable, use the `:=` operator as you would C or Visual

Basic's = operator. Pascal programmers often call this the *gets* or *assignment* operator, and the expression

```
Number1 := 5;
```

is read either "`Number1` *gets* the value `5`," or "`Number1` *is assigned* the value `5`."

Comparison Operators

If you've already programmed in Visual Basic, you should be very comfortable with Delphi's comparison operators because they're virtually identical. These operators are fairly standard throughout programming languages, so they're covered only briefly in this section.

Object Pascal uses the = operator to perform logical comparisons between two expressions or values. Object Pascal's = operator is analogous to C's == operator, so a C expression that would be written as

```
if (x == y)
```

would be written as this in Object Pascal:

```
if x = y
```

NOTE

Remember that in Object Pascal, the `:=` operator is used to assign a value to a variable, and the = operator compares the values of two operands.

Delphi's "not equal to" operator is <>, and its purpose is identical to C's `!=` operator. To determine whether two expressions are not equal, use this code:

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

Logical Operators

Pascal uses the words `and` and `or` as logical "and" and "or" operators, whereas C uses the `&&` and `||` symbols, respectively, for these operators. The most common use of the `and` and `or` operators is as part of an `if` statement or loop, as demonstrated in the following two examples:

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

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

Pascal's logical "not" operator is `not`, which is used to invert a Boolean expression. It's analogous to C's `!` operator. It's also often used as a part of `if` statements, as shown here:

```
if not (condition) then (do something);   // if condition is false then...
```

Table 2.1 provides an easy reference of how Pascal operators map to corresponding C/C++ and Visual Basic operators.

TABLE 2.1 Assignment, comparison, and logical operators

Operator	*Pascal*	*C/C++*	*Visual Basic*
Assignment	`:=`	`=`	`=`
Comparison	`=`	`==`	= or `Is`*
Not equal to	`<>`	`!=`	`<>`
Less than	`<`	`<`	`<`
Greater than	`>`	`>`	`>`
Less than or equal to	`<=`	`<=`	`<=`
Greater than or equal to	`>=`	`>=`	`>=`
Logical and	`and`	`&&`	`And`
Logical or	`or`	`||`	`Or`
Logical not	`not`	`!`	`Not`

**The `Is` comparison operator is used for objects, while the = comparison operator is used for other types.*

Arithmetic Operators

You should already be familiar with most Object Pascal arithmetic operators because they're generally similar to those used in C, C++, and Visual Basic. Table 2.2 illustrates all the Pascal arithmetic operators and their C/C++ and Visual Basic counterparts.

TABLE 2.2 Arithmetic operators

Operator	*Pascal*	*C/C++*	*Visual Basic*
Addition	`+`	`+`	`+`
Subtraction	`-`	`-`	`-`
Multiplication	`*`	`*`	`*`
Floating-point division	`/`	`/`	`/`
Integer division	`div`	`/`	`\`
Modulus	`mod`	`%`	`Mod`
Exponent	None	None	`^`

You may notice that Pascal and Visual Basic provide different division operators for floating-point and integer math, while this is not the case for C/C++. The `div` operator automatically truncates any remainder when you're dividing two integer expressions.

NOTE

Remember to use the correct division operator for the types of expressions with which you're working. The Object Pascal compiler gives you an error if you try to divide two floating-point numbers with the integer `div` operator or two integers with the floating-point `/` operator, as the following code illustrates:

```
var
  i: Integer;
  r: Real;
begin
  i := 4 / 3;         // This line will cause a compiler error
  f := 3.4 div 2.3;   // This line also will cause an error
end;
```

Many other programming languages do not distinguish between integer and floating-point division. Instead, they always perform floating-point division and then convert the result back to an integer when necessary. This can be rather expensive in terms of performance. The Pascal `div` operator is faster and more specific.

Bitwise Operators

Bitwise operators are operators that enable you to modify individual bits of a given variable. Common bitwise operators enable you to shift the bits to the left or right or to perform bitwise "and," "not," "or," and "exclusive or" *(xor)* operations with two numbers. The Shift+Left and Shift+Right operators are `shl` and `shr`, respectively, and they're much like C's << and >> operators. The remainder of Pascal's bitwise operators is easy enough to remember: `and`, `not`, `or`, and `xor`. Table 2.3 lists the bitwise operators.

TABLE 2.3 Bitwise operators

Operator	*Pascal*	*C*	*Visual Basic*
And	`and`	`&`	`And`
Not	`not`	`~`	`Not`
Or	`or`	`|`	`Or`
Xor	`xor`	`^`	`Xor`
Shift+Left	`shl`	`<<`	None
Shift+Right	`shr`	`>>`	None

Increment and Decrement Procedures

Increment and decrement procedures generate optimized code for adding or subtracting 1 from a given integral variable. Pascal doesn't really provide honest-to-gosh increment and decrement operators similar to C's `++` and `—-` operators, but Pascal's `Inc()` and `Dec()` procedures compile optimally to one machine instruction.

You can call `Inc()` or `Dec()` with one or two parameters. For example, the following two lines of code increment and decrement `variable`, respectively, by 1, using the `inc` and `dec` assembly instructions:

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

Compare the following two lines, which increment or decrement `variable` by 3 using the `add` and `sub` assembly instructions:

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

Table 2.4 compares the increment and decrement operators of different languages.

NOTE

With compiler optimization enabled, the `Inc()` and `Dec()` procedures often produce the same machine code as `variable :=variable + 1` syntax, so use whichever you feel more comfortable with for incrementing and decrementing variables.

TABLE 2.4 Increment and decrement operators

Operator	*Pascal*	*C*	*Visual Basic*
Increment	`Inc()`	`++`	None
Decrement	`Dec()`	`—-`	None

Object Pascal Types

One of Object Pascal's greatest features is that it's strongly typed, or *typesafe*. This means that actual variables passed to procedures and functions must be of the same type as the formal parameters identified in the procedure or function definition. You won't see any of the famous compiler warnings about suspicious pointer conversions that C programmers have grown to know and love. This is because the Object Pascal compiler will not permit you to call a function with one type of pointer when another type is specified in the function's formal parameters (although functions that take untyped `Pointer` types accept any type of pointer). Basically, Pascal's strongly typed nature enables it to perform a sanity check of your code—to ensure you're not trying to put a square peg in a round hole.

A Comparison of Types

Delphi's base types are similar to those of C and Visual Basic. Table 2.5 compares and contrasts the base types of Object Pascal with those of C/C++ and Visual Basic. You may want to earmark this page because this table provides an excellent reference for matching types when calling functions in non-Delphi *dynamic link libraries* (DLLs) or *object files* (OBJs) from Delphi (and vice versa).

TABLE 2.5 A PASCAL TO C/C++ TO VISUAL BASIC 32-BIT TYPE COMPARISON

Type of Variable	*Pascal*	*C/C++*	*Visual Basic*
8-bit signed integer	`ShortInt`	`char`	None
8-bit unsigned integer	`Byte`	`BYTE`, `unsigned short`	Byte
16-bit signed integer	`SmallInt`	`short`	`Short`
16-bit unsigned integer	`Word`	`unsigned short`	None
32-bit signed integer	`Integer`, `Longint`	`int`, `long`	`Integer`, `Long`
32-bit unsigned integer	`Cardinal`, `LongWord`	`unsigned long`	None
64-bit signed integer	`Int64`	`__int64`	None
4-byte floating point	`Single`	`float`	`Single`
6-byte floating point	`Real48`	None	None
8-byte floating point	`Double`	`double`	`Double`
10-byte floating point	`Extended`	`long double`	None
64-bit currency	`currency`	None	`Currency`
8-byte date/time	`TDateTime`	None	`Date`
16-byte variant	`Variant`, `OleVariant`, `TVarData`	`VARIANT` `Variant`†, `OleVariant`†	`Variant` (Default)
1-byte character	`Char`	`char`	None

continues

Table 2.5 Continued

Type of Variable	*Pascal*	*C/C++*	*Visual Basic*
2-byte character	`WideChar`	`WCHAR`	
Fixed-length byte string	`ShortString`	None	None
Dynamic string	`AnsiString`	`AnsiString`†	String
Null-terminated string	`PChar`	`char *`	None
Null-terminated wide string	`PWideChar`	`LPCWSTR`	None
Dynamic 2-byte string	`WideString`	`WideString`†	None
1-byte Boolean	`Boolean`, `ByteBool`	(Any 1-byte)	None
2-byte Boolean	`WordBool`	(Any 2-byte)	`Boolean`
4-byte Boolean	BOOL, LongBool	BOOL	None

† *A Borland C++Builder class that emulates the corresponding Object Pascal type*

Note

If you're porting 16-bit code from Delphi 1.0, be sure to bear in mind that the size of both the `Integer` and `Cardinal` types has increased from 16 to 32 bits. Actually, that's not quite accurate: Under Delphi 2 and 3 the `Cardinal` type was treated as an unsigned 31-bit integer in order to preserve arithmetic precision (because Delphi 2 and 3 lacked a true unsigned 32-bit integer to which results of integer operations could be promoted). Under Delphi 4 and higher, `Cardinal` is a true unsigned 32-bit integer.

Caution

In Delphi 1, 2, and 3, the `Real` type identifier specified a 6-byte floating-point number, which is a type unique to Pascal and generally incompatible with other languages. In Delphi 4, `Real` is an alias for the `Double` type. The old 6-byte floating-point number is still there, but it's now identified by `Real48`. You can also force the `Real` identifier to refer to the 6-byte floating-point number using the `{$REALCOMPATIBILITY ON}` directive.

Characters

Delphi provides three character types:

- `AnsiChar`. This is the standard one-byte ANSI character that programmers have grown to know and love.
- `WideChar`. This character is two bytes in size and represents a Unicode character.
- `Char`. This is currently identical to `AnsiChar`, but Borland warns that the definition may change in a later version of Delphi to a `WideChar`.

Keep in mind that because a character is no longer guaranteed to be one byte in size, you shouldn't hard-code the size into your applications. Instead, you should use the `SizeOf()` function where appropriate.

NOTE

The `SizeOf()` standard procedure returns the size, in bytes, of a type or instance.

A Multitude of Strings

Strings are variable types used to represent groups of characters. Every language has its own spin on how string types are stored and used. Pascal has several different string types to suit your programming needs:

- `AnsiString`, the default string type for Object Pascal, is comprised of `AnsiChar` characters and allows for virtually unlimited lengths. It's also compatible with null-terminated strings.
- `ShortString` remains in the language primarily for backward compatibility with Delphi 1. Its capacity is limited to 255 characters.
- `WideString` is similar in functionality to `AnsiString` except that it consists of `WideChar` characters.
- `PChar` is a pointer to a null-terminated `Char` string—like C's `char *` and `lpstr` types.
- `PAnsiChar` is a pointer to a null-terminated `AnsiChar` string.
- `PWideChar` is a pointer to a null-terminated `WideChar` string.

By default, when you declare a `string` variable in your code, as shown in the following example, the compiler assumes that you're creating an `AnsiString`:

```
var
  S: string;   // S is an AnsiString
```

Alternatively, you can cause variables declared as `string` types to instead be of type `ShortString` using the `$H` compiler directive. When the value of the `$H` compiler directive is negative, `string` variables are `ShortString` types, and when the value of the directive is positive (the default), `string` variables are `AnsiString` types. The following code demonstrates this behavior:

```
var
  {$H-}
  S1: string;  // S1 is a ShortString
  {$H+}
  S2: string;  // S2 is an AnsiString
```

The exception to the `$H` rule is that a `string` declared with an explicit size (limited to a maximum of 255 characters) is always a `ShortString`:

```
var
  S: string[63];    // A ShortString of up to 63 characters
```

The AnsiString Type

The `AnsiString` (or *long string)* type was introduced to the language in Delphi 2. It exists primarily as a result of widespread Delphi 1 customer demand for an easy-to-use string type without the intrusive 255-character limitation. `AnsiString` is that and more.

Although `AnsiString` types maintain an interface almost identical their predecessors, they're dynamically allocated and garbage-collected. Because of this, `AnsiString` is sometimes referred to as a *lifetime-managed* type. Object Pascal also automatically manages allocation of string temporaries as needed, so you needn't worry about allocating buffers for intermediate results as you would in C/C++. Additionally, `AnsiString` types are always guaranteed to be null terminated, which makes them compatible with the null-terminated strings used by the Win32 API. The `AnsiString` type is actually implemented as a pointer to a string structure in heap memory. Figure 2.1 shows how an `AnsiString` is laid out in memory.

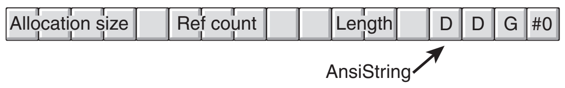


FIGURE 2.1
An `AnsiString` in memory.

CAUTION

The complete internal format of the long string type is left undocumented by Borland, and Borland reserves the right to change the internal format of long strings with future releases of Delphi. The information here is intended mainly to help you understand how `AnsiString` types work, and you should avoid being dependent on the structure of an `AnsiString` in your code.

Developers who avoided the implementation of details of string moving from Delphi 1 to Delphi 2 were able to migrate their code with no problems. Those who wrote code that depended on the internal format (such as the 0th element in the string being the length) had to modify their code for Delphi 2.

As Figure 2.1 illustrates, `AnsiString` types are reference counted, which means that several strings may point to the same physical memory. String copies, therefore, are very fast because it's merely a matter of copying a pointer rather than copying the actual string contents. When two or more `AnsiString` types share a reference to the same physical string, the Delphi memory manager uses a copy-on-write technique, which enables it to wait until a string is modified to release a reference and allocate a new physical string. The following example illustrates these concepts:

```
var
  S1, S2: string;
begin
  // store string in S1, ref count of S1 is 1
  S1 := 'And now for something... ';
  S2 := S1;          // S2 now references S1.  Ref count of S1 is 2.
  // S2 is changed, so it is copied to its own
  // memory space, and ref count of S1 is decremented



  S2 := S2 + 'completely different!';
```

Lifetime-Managed Types

In addition to `AnsiString`, Delphi provides several other types that are lifetime-managed. These types include `WideString`, `Variant`, `OleVariant`, `interface`, `dispinterface`, and dynamic arrays. You'll learn more about each of these types later in this chapter. For now, we'll focus on what exactly lifetime-managed types are and how they work.

Lifetime-managed types, sometimes called *garbage-collected types*, are types that potentially consume some particular resource while in use and release the resource automatically when they fall out of scope. Of course, the variety of resources used

continues

depends on the type involved. For example, an `AnsiString` consumes memory for the character string while in use, and the memory occupied by the character string is released when it leaves scope.

For global variables, this process is fairly straightforward: As a part of the finalization code generated for your application, the compiler inserts code to ensure that each lifetime-managed global variable is cleaned up. Because all global data is zero-initialized when your application loads, each lifetime-managed global variable will always initially contain a zero, empty, or some other value indicating the variable is "unused." This way, the finalization code won't attempt to free resources unless they're actually used in your application.

Whenever you declare a local lifetime-managed variable, the process is slightly more complex: First, the compiler inserts code to ensure that the variable is initialized to zero when the function or procedure is entered. Next, the compiler generates a `try..finally` exception-handling block, which it wraps around the entire function body. Finally, the compiler inserts code in the `finally` block to clean up the lifetime-managed variable (exception handling is explained in more detail in the section "Structured Exception Handling"). With this in mind, consider the following procedure:

```
procedure Foo;
var
  S: string;
begin
  // procedure body
  // use S here
end;
```

Although this procedure looks simple, if you take into account the code generation by the compiler behind the scenes, it would actually look like this:

```
procedure Foo;
var
  S: string;
begin
  S := '';
  try
    // procedure body
    // use S here
  finally
    // clean up S here
  end;
end;
```

String Operations

You can concatenate two `strings` by using the `+` operator or the `Concat()` function. The preferred method of string concatenation is the `+` operator because the `Concat()` function exists primarily for backward compatibility. The following example demonstrates the use of + and `Concat()`:

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

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

NOTE

Always use single quotation marks (`'A String'`) when working with string literals in Object Pascal.

TIP

`Concat()` is one of many “compiler magic” functions and procedures (like `ReadLn()` and `WriteLn()`, for example) that don’t have an Object Pascal definition. Because such functions and procedures are intended to accept an indeterminate number of parameters or optional parameters, they cannot be defined in terms of the Object Pascal language. Because of this, the compiler provides a special case for each of these functions and generates a call to one of the “compiler magic” *helper functions* defined in the `System` unit. These helper functions are generally implemented in assembly language in order to circumvent Pascal language rules.

In addition to the “compiler magic” string support functions and procedures, there are a variety of functions and procedures in the `SysUtils` unit designed to make working with strings easier. Search for “String-handling routines (Pascal-style)” in the Delphi online help system.

Furthermore, you’ll find some very useful homebrewed string utility functions and procedures in the `StrUtils` unit in the `\Source\Utils` directory on the CD-ROM accompanying this book.

Length and Allocation

When first declared, an `AnsiString` has no length and therefore no space allocated for the characters in the string. To cause space to be allocated for the string, you can assign the string to a literal or another string, or you can use the `SetLength()` procedure, as shown here:

```
var
  S: string;        // string initially has no length
begin
  S := 'Doh!';      // allocates at least enough space for string literal
  { or }
  S := OtherString  // increases ref count of OtherString
                    // (assume OtherString already points to a valid string)
  { or }
  SetLength(S, 4);  // allocates enough space for at least 4 chars
end;
```

You can index the characters of an `AnsiString` like an array, but be careful not to index beyond the length of the string. For example, the following code snippet will cause an error:

```
var
  S: string;
begin
  S[1] := 'a';  // Won't work because S hasn't been allocated!
end;
```

This code, however, works properly:

```
var
  S: string;
begin
  SetLength(S, 1);
  S[1] := 'a';      // Now S has enough space to hold the character
end;
```

Win32 Compatibility

As mentioned earlier, `AnsiString` types are always null-terminated, so they're compatible with null-terminated strings. This makes it easy to call Win32 API functions or other functions requiring `PChar`-type strings. All that's required is that you typecast the `string` as a `PChar` (typecasting is explained in more detail in the section "Typecasting and Type Conversion"). The following code demonstrates how to call the Win32 `GetWindowsDirectory()` function, which accepts a `PChar` and buffer length as parameters:

```
var
  S: string;
begin
  SetLength(S, 256);           // important! get space for string first
  // call function, S now holds directory string
  GetWindowsDirectory(PChar(S), 256);
end;
```

After using an `AnsiString` where a function or procedure expects a `PChar`, you must manually set the length of the string variable to its null-terminated length. The `RealizeLength()` function, which also comes from the `STRUTILS` unit, accomplishes that task:

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

Calling `RealizeLength()` completes the substitution of a long string for a `PChar`:

```
var
  S: string;
begin
  SetLength(S, 256);                  // important! get space for string first
  // call function, S now holds directory string
  GetWindowsDirectory(PChar(S), 256);
  RealizeLength(S);          // set S length to null length
end;
```

CAUTION

Exercise care when typecasting a `string` to a `PChar` variable. Because strings are garbage-collected when they go out of scope, you must pay attention when making assignments such as `P := PChar(Str)`, where the scope (or lifetime) of `P` is greater than `Str`.

Porting Issues

When you're porting 16-bit Delphi 1 applications, you need to keep in mind a number of issues when migrating to `AnsiString` types:

- In places where you used the `PString` (pointer to a `ShortString`) type, you should instead use the `string` type. Remember, an `AnsiString` is already a pointer to a string.
- You can no longer access the 0th element of a string to get or set the length. Instead, use the `Length()` function to get the string length and the `SetLength()` procedure to set the length.
- There's no longer any need to use `StrPas()` and `StrPCopy()` to convert back and forth between strings and `PChar` types. As shown earlier, you can typecast an `AnsiString` to a `PChar`. When you want to copy the contents of a `PChar` to an `AnsiString`, you can use a direct assignment:

  ```
  StringVar := PCharVar;
  ```

CAUTION

Remember that you must use the `SetLength()` procedure to set the length of a long string, whereas the past practice was to directly access the 0th element of a short string to set the length. This issue will arise when you attempt to port 16-bit Delphi 1.0 code to 32 bits.

The ShortString Type

If you're a Delphi veteran, you'll recognize the `ShortString` type as the Delphi 1.0 `string` type. `ShortString` types are sometimes referred to as *Pascal strings* or *length-byte strings*. To reiterate, remember that the value of the `$H` directive determines whether variables declared as `string` are treated by the compiler as `AnsiString` or `ShortString`.

In memory, the string resembles an array of characters where the 0th character in the string contains the length of the string, and the string itself is contained in the following characters. The storage size of a `ShortString` defaults to the maximum of 256 bytes. This means that you can never have more than 255 characters in a `ShortString` (255 characters + 1 length byte = 256). As with `AnsiString`, working with `ShortString` is fairly painless because the compiler allocates string temporaries as needed, so you don't have to worry about allocating buffers for intermediate results or disposing of them as you do with C.

Figure 2.2 illustrates how a Pascal string is laid out in memory.

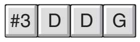


FIGURE 2.2
A `ShortString` in memory.

A `ShortString` variable is declared and initialized with the following syntax:

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

Optionally, you can allocate fewer than 256 bytes for a `ShortString` using just the `string` type identifier and a length specifier, as in the following example:

```
var
  S: string[45];  { a 45-character ShortString }
begin
  S := 'This string must be 45 or fewer characters.';
end.
```

The preceding code causes a ShortString to be created regardless of the current setting of the $H directive. The maximum length you can specify is 255 characters.

Never store more characters to a ShortString than you have allocated memory for. If you declare a variable as a string[8], for example, and try to assign 'a_pretty_darn_long_string' to that variable, the string would be truncated to only eight characters, and you would lose data.

When using an array subscript to address a particular character in a ShortString, you could get bogus results or corrupt memory if you attempt to use a subscript index that's greater than the declared size of the ShortString. For example, suppose you declare a variable as follows:

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

If you then attempt to write to the 10th element of the string as follows, you're likely to corrupt memory used by other variables:

```
var
  Str: string[8];
  i: Integer;
begin
  i := 10;
  Str[i] := 's';  // will corrupt memory
```

You can have the compiler link in special logic catch these types of errors at runtime by selecting Range Checking in the Options, Project dialog box.

TIP

Although including range-checking logic in your program helps you find string errors, range checking slightly hampers the performance of your application. It's common practice to use range checking during the development and debugging phases of your program, but you should remove range checking after you become confident in the stability of your program.

Unlike AnsiString types, ShortString types are not inherently compatible with null-terminated strings. Because of this, a bit of work is required to be able to pass a ShortString to a Win32 API function. The following function, ShortStringAsPChar(), is taken from the STRUTILS.PAS unit mentioned earlier:

```
func function ShortStringAsPChar(var S: ShortString): PChar;
{ Function null-terminates a string so it can be passed to functions }
{ that require PChar types. If string is longer than 254 chars, then it will   }
{ be truncated to 254. }
begin
  if Length(S) = High(S) then Dec(S[0]); { Truncate S if it's too long }
  S[Ord(Length(S)) + 1] := #0;           { Place null at end of string }
  Result := @S[1];                       { Return "PChar'd" string }
end;
```

CAUTION

The functions and procedures in the Win32 API require null-terminated strings. Do not try to pass a `ShortString` type to an API function because your program will not compile. Your life will be easier if you use long strings when working with the API.

The WideString Type

The `WideString` type is a lifetime-managed type similar to `AnsiString`; they're both dynamically allocated, garbage collected, and even assignment compatible with one another. However, `WideString` differs from `AnsiString` in three key respects:

- `WideString` types consist of `WideChar` characters rather than `AnsiChar` characters, making them compatible with Unicode strings.
- `WideString` types are allocated using the `SysAllocStrLen()` API function, making them compatible with OLE `BSTR` strings.
- `WideString` types are not reference counted, so assigning one `WideString` to another requires the entire string to be copied from one location in memory to another. This makes `WideString` types less efficient than `AnsiString` types in terms of speed and memory use.

As mentioned earlier, the compiler automatically knows how to convert between variables of `AnsiString` and `WideString` types, as shown here:

```
var
  W: WideString;
  S: string;
begin
  W := 'Margaritaville';
  S := W;  // Wide converted to Ansi
  S := 'Come Monday';
  W := S;  // Ansi converted to Wide
end;
```

In order to make working with `WideString` types feel natural, Object Pascal overloads the `Concat()`, `Copy()`, `Insert()`, `Length()`, `Pos()`, and `SetLength()` routines and the `+`, `=`, and `<>` operators for use with `WideString` types. Therefore, the following code is syntactically correct:

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

As with the `AnsiString` and `ShortString` types, you can use array brackets to reference individual characters of a `WideString`:

```
var
  W: WideString;
  C: WideChar;
begin
  W := 'Ebony and Ivory living in perfect harmony';
  C := W[Length(W)];  // C holds the last character in W
end;
```

Null-Terminated Strings

Earlier, this chapter mentioned that Delphi has three different null-terminated string types: `PChar`, `PAnsiChar`, and `PWideChar`. As their names imply, each of these represents a null-terminated string of each of Delphi's three character types. In this chapter, we refer to each of these string types generically as `PChar`. The `PChar` type in Delphi exists mainly for compatibility with Delphi 1.0 and the Win32 API, which makes extensive use of null-terminated strings. A `PChar` is defined as a pointer to a string followed by a null (zero) value (if you're unsure of exactly what a pointer is, read on; pointers are discussed in more detail later in this section). Unlike memory for `AnsiString` and `WideString` types, memory for `PChar` types is not automatically allocated and managed by Object Pascal. Therefore, you'll usually need to allocate memory for the string to which it points, using one of Object Pascal's memory-allocation functions. The theoretical maximum length of a `PChar` string is just under 4GB. The layout of a `PChar` variable in memory is shown in Figure 2.3.

TIP

Because Object Pascal's `AnsiString` type can be used as a `PChar` in most situations, you should use this type rather than the `PChar` type wherever possible. Because memory management for strings occurs automatically, you greatly reduce the chance of introducing memory-corruption bugs into your applications if, where possible, you avoid `PChar` types and the manual memory allocation associated with them.

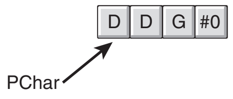


FIGURE 2.3
A PChar in memory.

As mentioned earlier, `PChar` variables require you to manually allocate and free the memory buffers that contain their strings. Normally, you allocate memory for a `PChar` buffer using the `StrAlloc()` function, but several other functions can be used to allocate memory for `PChar` types, including `AllocMem()`, `GetMem()`, `StrNew()`, and even the `VirtualAlloc()` API function. Corresponding functions also exist for many of these functions, which must be used to deallocate memory. Table 2.6 lists several allocation functions and their corresponding deallocation functions.

TABLE 2.6 Memory allocation and deallocation functions

Memory Allocated with…	*Must Be Freed with…*
`AllocMem()`	`FreeMem()`
`GlobalAlloc()`	`GlobalFree()`
`GetMem()`	`FreeMem()`
`New()`	`Dispose()`
`StrAlloc()`	`StrDispose()`
`StrNew()`	`StrDispose()`
`VirtualAlloc()`	`VirtualFree()`

The following example demonstrates memory allocation techniques when working with `PChar` and `string` types:

```
var
  P1, P2: PChar;
  S1, S2: string;
begin
  P1 := StrAlloc(64 * SizeOf(Char));  // P1 points to an allocation of 63 Chars
  StrPCopy(P1, 'Delphi 5 ');          // Copy literal string into P1
  S1 := 'Developer''s Guide';         // Put some text in string S1
  P2 := StrNew(PChar(S1));            // P1 points to a copy of S1
  StrCat(P1, P2);                     // concatenate P1 and P2
  S2 := P1;                       // S2 now holds 'Delphi 5 Developer's Guide'
  StrDispose(P1);                     // clean up P1 and P2 buffers
  StrDispose(P2);
```

```
end.
```

Notice, first of all, the use of `SizeOf(Char)` with `StrAlloc()` when allocating memory for `P1`. Remember that the size of a `Char` may change from one byte to two in future versions of Delphi; therefore, you cannot assume the value of `Char` to always be one byte. `SizeOf()` ensures that the allocation will work properly no matter how many bytes a character occupies.

`StrCat()` is used to concatenate two `PChar` strings. Note here that you cannot use the `+` operator for concatenation as you can with long string and `ShortString` types.

The `StrNew()` function is used to copy the value contained by string `S1` into `P2` (a `PChar`). Be careful when using this function. It's common to have memory-overwrite errors when using `StrNew()` because it allocates only enough memory to hold the string. Consider the following example:

```
var
P1, P2: Pchar;
begin
  P1 := StrNew('Hello ');  // Allocate just enoughmemory for P1 and P2
  P2 := StrNew('World');
  StrCat(P1, P2);          // BEWARE: Corrupts memory!
  .
```

TIP

```
  .
  .
end;
```

As with other types of strings, Object Pascal provides a decent library of utility functions and procedures for operating on `PChar` types. Search for "String-handling routines (null-terminated)" in the Delphi online help system.

You'll also find some useful null-terminated functions and procedures in the `StrUtils` unit in the `\Source\Utils` directory on the CD-ROM accompanying this book.

Variant Types

Delphi 2.0 introduced a powerful data type called the `Variant`. Variants were brought about primarily in order to support OLE Automation, which uses the `Variant` type heavily. In fact, Delphi's `Variant` data type is an encapsulation of the variant used with OLE. Delphi's implementation of variants has also proven to be useful in other areas of Delphi programming, as you'll soon learn. Object Pascal is the only compiled language that completely integrates variants as a dynamic data type at runtime and as a static type at compile time in that the compiler always knows that it's a variant.

Delphi 3 introduced a new type called `OleVariant`, which is identical to `Variant` except that it can only hold Automation-compatible types. In this section, we initially focus on the `Variant` type and then we discuss `OleVariant` and contrast it with `Variant`.

Variants Change Types Dynamically

One of the main purposes of variants is to have a variable whose underlying data type cannot be determined at compile time. This means that a variant can change the type to which it refers at runtime. For example, the following code will compile and run properly:

```
var
  V: Variant;
begin
  V := 'Delphi is Great!';   // Variant holds a string
  V := 1;                      // Variant now holds an Integer
  V := 123.34;                 // Variant now holds a floating point
  V := True;                   // Variant now holds a boolean
  V := CreateOleObject('Word.Basic'); // Variant now holds an OLE object
end;
```

Variants can support all simple data types, such as integers, floating-point values, strings, Booleans, date and time, currency, and also OLE Automation objects. Note that variants cannot refer to Object Pascal objects. Also, variants can refer to a nonhomogeneous array, which can vary in size and whose data elements can refer to any of the preceding data types (including another variant array).

The Variant Structure

The data structure defining the `Variant` type is defined in the `System` unit and is also shown in the following code:

```
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;
```

The `TVarData` structure consumes 16 bytes of memory. The first two bytes of the `TVarData` structure contain a word value that represents the data type to which the variant refers. The following code shows the various values that may appear in the `VType` field of the `TVarData` record. The next six bytes are unused. The remaining eight bytes contain the actual data or a pointer to the data represented by the variant. Again, this structure maps directly to OLE's implementation of the variant type. Here's the code:

```
{ Variant type codes }
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;
```

NOTE

```
varString   = $0100;
varAny      = $0101;
```

```
  varTypeMask = $0FFF;
  varArray    = $2000;
  varByRef    = $4000;
```

As you may notice from the type codes in the preceding listing, a `Variant` cannot contain a reference to a `Pointer` or `class` type.

You'll notice from the `TVarData` listing that the `TVarData` record is actually a *variant record*. Don't confuse this with the `Variant` type. Although the variant record and `Variant` type have similar names, they represent two totally different constructs. Variant records allow for multiple data fields to overlap in the same area of memory (like a C/C++ union). This is discussed in more detail in the "Records" section later in this chapter. The `case` statement in the `TVarData` variant record indicates the type of data to which the variant refers. For example, if the `VType` field contains the value `varInteger`, only four bytes of the eight data bytes in the variant portion of the record are used to hold an integer value. Likewise, if `VType` has the value `varByte`, only one byte of the eight are used to hold a byte value.

You'll notice that if `VType` contains the value `varString`, the eight data bytes don't actually hold the string; instead, they hold a pointer to this string. This is an important point because you can access fields of a variant directly, as shown here:

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

You must understand that in some cases this is a dangerous practice because it's possible to lose the reference to a string or other lifetime-managed entity, which will result in your application leaking memory or other resources. You'll see what we mean by the term *garbage collected* in the following section.

Variants Are Lifetime Managed

Delphi automatically handles the allocation and deallocation of memory required of a `Variant` type. For example, examine the following code, which assigns a string to a `Variant` variable:

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

As discussed earlier in this chapter in the sidebar on lifetime-managed types, several things are going on here that might not be apparent. Delphi first initializes the variant to an unassigned value. During the assignment, it sets its `VType` field to `varString` and copies the string pointer into its `VString` field. It then increases the reference count of string `S`. When the variant leaves scope (that is, the procedure ends and returns to the code that called it), it's cleared and the reference count of string `S` is decremented. Delphi does this by implicitly inserting a `try..finally` block in the procedure, as shown here:

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := Unassigned;  // initialize variant to "empty"
  try
    V := S;
    ShowMessage(V);
  finally
    // Now clean up the resources associated with the variant
  end;
end;
```

This same implicit release of resources occurs when you assign a different data type to the variant. For example, examine the following code:

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

This code boils down to the following pseudo-code:

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  Clear Variant V, ensuring it is initialized to "empty"
  try
    V.VType := varString; V.VString := S; Inc(S.RefCount);
    Clear Variant V, thereby releasing reference to string;
    V.VType := varInteger; V.VInteger := 34;
  finally
    Clean up the resources associated with the variant
  end;
end;
```

If you understand what happens in the preceding examples, you'll see why it's not recommended that you manipulate fields of the `TVarData` record directly, as shown here:

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

Although this may appear to be safe, it's not because it results in the failure to decrement the reference count of string `S`, probably resulting in a memory leak. As a general rule, don't access the `TVarData` fields directly, or if you do, be absolutely sure that you know exactly what you're doing.

Typecasting Variants

You can explicitly typecast expressions to type `Variant`. For example, the expression

```
Variant(X)
```

results in a `Variant` type whose type code corresponds to the result of the expression `X`, which must be an integer, real, currency, string, character, or Boolean type.

You can also typecast a variant to that of a simple data type. For example, given the assignment

```
V := 1.6;
```

where `V` is a variable of type `Variant`, the following expressions will have the results shown:

```
S := string(V);    // S will contain the string '1.6';
// I is rounded to the nearest Integer value, in this case: 2.
I := Integer(V);
B := Boolean(V);   // B contains False if V contains 0, otherwise B is True
D := Double(V);    // D contains the value 1.6
```

These results are dictated by certain type-conversion rules applicable to `Variant` types. These rules are defined in detail in Delphi's Object Pascal Language Guide.

By the way, in the preceding example, it's not necessary to typecast the variant to another data type to make the assignment. The following code would work just as well:

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

What happens here is that the conversions to the target data types are made through an implicit typecast. However, because these conversions are made at runtime, there's much more code logic attached to this method. If you're sure of the type a variant contains, you're better off explicitly typecasting it to that type in order to speed up the operation. This is especially true if the variant is being used in an expression, which we'll discuss next.

Variants in Expressions

You can use variants in expressions with the following operators: `+`, `=`, `*`, `/`, `div`, `mod`, `shl`, `shr`, `and`, `or`, `xor`, `not`, `:=`, `<>`, `<`, `>`, `<=`, and `>=`.

When using variants in expressions, Delphi knows how to perform the operations based on the contents of the variant. For example, if two variants, `V1` and `V2`, contain integers, the expression `V1 + V2` results in the addition of the two integers. However, if `V1` and `V2` contain strings, the result is a concatenation of the two strings. What happens if `V1` and `V2` contain two different data types? Delphi uses certain promotion rules in order to perform the operation. For example, if `V1` contains the string `'4.5'` and `V2` contains a floating-point number, `V1` will be converted to a floating point and then added to `V2`. The following code illustrates this:

```
var
  V1, V2, V3: Variant;
begin
  V1 := '100';  // A string type
  V2 := '50';   // A string type
  V3 := 200;    // An Integer type
  V1 := V1 + V2 + V3;
end;
```

Based on what we just mentioned about promotion rules, it would seem at first glance that the preceding code would result in the value `350` as an integer. However, if you take a closer look, you'll see that this is not the case. Because the order of precedence is from left to right, the first equation executed is `V1 + V2`. Because these two variants refer to strings, a string concatenation is performed, resulting in the string `'10050'`. That result is then added to the integer value held by the variant `V3`. Because `V3` is an integer, the result `'10050'` is converted to an integer and added to `V3`, thus providing an end result of `10250`.

Delphi promotes the variants to the highest type in the equation in order to successfully carry out the calculation. However, when an operation is attempted on two variants of which Delphi cannot make any sense, an "invalid variant type conversion" exception is raised. The following code illustrates this:

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'hello';
  V1 := V1 / V2;  // Raises an exception.
```

```
end;
```

As stated earlier, it's sometimes a good idea to explicitly typecast a variant to a specific data type if you know what that type is and if it's used in an expression. Consider the following line of code:

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

Before a result can be generated for this equation, each operation is handled by a runtime function that goes through several gyrations to determine the compatibility of the types the variants represent. Then the conversions are made to the appropriate data types. This results in a large amount of overhead and code size. A better solution is obviously not to use variants. However, when necessary, you can also explicitly typecast the variants so the data types are resolved at compile time:

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

Keep in mind that this assumes you know the data types the variants represent.

Empty and Null

Two special `VType` values for variants merit a brief discussion. The first is `varEmpty`, which means that the variant has not yet been assigned a value. This is the initial value of the variant set by the compiler as it comes into scope. The other is `varNull`, which is different from `varEmpty` in that it actually represents the value `Null` as opposed to a lack of value. This distinction between no value and a `Null` value is especially important when applied to the field values of a database table. In Chapter 27, "Writing Desktop Database Applications," you'll learn how variants are used in the context of database applications.

Another difference is that attempting to perform any equation with a variant containing a `varEmpty` `VType` value will result in an "invalid variant operation" exception. The same is not

CAUTION

true of variants containing a `varNull` value, however. When a variant involved in an equation contains a `Null` value, that value will propagate to the result. Therefore, the result of any equation containing a `Null` is always `Null`.

If you want to assign or compare a variant to one of these two special values, the `System` unit defines two variants, `Unassigned` and `Null`, which have the `VType` values of `varEmpty` and `varNull`, respectively.

It may be tempting to use variants instead of the conventional data types because they seem to offer so much flexibility. However, this will increase the size of your code and cause your applications to run more slowly. Additionally, it will make your code more difficult to maintain. Variants are useful in many situations. In fact, the VCL, itself, uses variants in several places, most notably in the ActiveX and database areas, because of the data type flexibility they offer. Generally speaking, however, you should use the conventional data types instead of variants. Only in situations where the flexibility of the variant outweighs the performance of the conventional method should you resort to using variants. Ambiguous data types beget ambiguous bugs.

Variant Arrays

Earlier we mentioned that a variant can refer to a nonhomogeneous array. Therefore, the following syntax is valid:

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

Bear in mind that, although the preceding code will compile, you'll get an exception at runtime because `V` does not yet contain a variant array. Object Pascal provides several variant array support functions that allow you to create a variant array. Two of these functions are `VarArrayCreate()` and `VarArrayOf()`.

VarArrayCreate()

`VarArrayCreate()` is defined in the `System` unit as

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

To use `VarArrayCreate()`, you pass in the array bounds for the array you want to create and a variant type code for the type of the array elements (the first parameter is an open array, which is discussed in the "Passing Parameters" section later in this chapter). For example, the following code returns a variant array of integers and assigns values to the array items:

```
var
  V: Variant;
begin
  V := VarArrayCreate([1, 4], varInteger); // Create a 4-element array
  V[1] := 1;
  V[2] := 2;
  V[3] := 3;
  V[4] := 4;
```

```
end;
```

If variant arrays of a single type aren't confusing enough, you can pass `varVariant` as the type code in order to create a variant array of variants! This way, each element in the array has the ability to contain a different type of data. You can also create a multidimensional array by passing in the additional bounds required. For example, the following code creates an array with the bounds `[1..4, 1..5]`:

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

VarArrayOf()

The `VarArrayOf()` function is defined in the `System` unit as

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

This function returns a one-dimensional array whose elements are given in the `Values` parameter. The following example creates a variant array of three elements with an integer, a string, and a floating-point value:

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

Variant Array Support Functions and Procedures

In addition to `VarArrayCreate()` and `VarArrayOf()`, there are several other variant array support functions and procedures. These functions are defined in the `System` unit and are also shown here:

```
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;
```

The `VarArrayRedim()` function allows you to resize the upper bound of the rightmost dimension of a variant array. The `VarArrayDimCount()` function returns the number of dimensions in a variant array. `VarArrayLowBound()` and `VarArrayHighBound()` return the lower and upper bounds of an array, respectively. `VarArrayLock()` and `VarArrayUnlock()` are two special functions, which are described in later detail in the next section.

`VarArrayRef()` is intended to work around a problem that exists in passing variant arrays to OLE Automation servers. The problem occurs when you pass a variant containing a variant array to an automation method, like this:

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

The array is passed not as a variant array but rather as a variant containing a variant array—an important distinction. If the server expected a variant array rather than a reference to one, the server will likely encounter an error condition when you call the method with the preceding syntax. `VarArrayRef()` takes care of this situation by massaging the variant into the type and value expected by the server. Here's the syntax for using `VarArrayRef()`:

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

`VarIsArray()` is a simple Boolean check, which returns `True` if the variant parameter passed to it is a variant array or `False` otherwise.

Initializing a Large Array: VarArrayLock() and VarArrayUnlock()

Variant arrays are important in OLE Automation because they provide the only means for passing raw binary data to an OLE Automation server (note that pointers are not a legal type in OLE Automation, as you'll learn in Chapter 23, "COM and ActiveX"). However, if used incorrectly, variant arrays can be a rather inefficient means of exchanging data. Consider the following line of code:

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

This line creates a variant array of 10,000 bytes. Suppose you have another array (nonvariant) declared of the same size and you want to copy the contents of this nonvariant array to the variant array. Normally, you can only do this by looping through the elements and assigning them to the elements of the variant array, as shown here:

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

The problem with this code is that it's bogged down by the significant overhead required just to initialize the variant array elements. This is due to the assignments to the array elements having to go through the runtime logic to determine type compatibility, the location of each element, and so forth. To avoid these runtime checks, you can use the `VarArrayLock()` function and the `VarArrayUnlock()` procedure.

`VarArrayLock()` locks the array in memory so that it cannot be moved or resized while it's locked, and it returns a pointer to the array data. `VarArrayUnlock()` unlocks an array locked with `VarArrayLock()` and once again allows the variant array to be resized and moved in memory. After the array is locked, you can employ a more efficient means to initialize the data by using, for example, the `Move()` procedure with the pointer to the array's data. The following code performs the initialization of the variant array shown earlier, but in a much more efficient manner:

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

Supporting Functions

There are several other support functions for variants that you can use. These functions are declared in the `System` unit and are also listed here:

```
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;
```

The `VarClear()` procedure clears a variant and sets the `VType` field to `varEmpty`. `VarCopy()` copies the `Source` variant to the `Dest` variant. The `VarCast()` procedure converts a variant to a specified type and stores that result into another variant. `VarType()` returns one of the `var`*XXX* type codes for a specified variant. `VarAsType()` has the same functionality as `VarCast()`. `VarIsEmpty()` returns `True` if the type code on a specified variant is `varEmpty`. `VarIsNull()` indicates whether a variant contains a `Null` value. `VarToStr()` converts a variant to its string representation (an empty string in the case of a `Null` or empty variant). `VarFromDateTime()` returns a variant that contains a given `TDateTime` value. Finally, `VarToDateTime()` returns the `TDateTime` value contained in a variant.

`OleVariant`

The `OleVariant` type is nearly identical to the `Variant` type described throughout this section of this chapter. The only difference between `OleVariant` and `Variant` is that `OleVariant` only supports Automation-compatible types. Currently, the only `VType` supported that's not Automation-compatible is `varString`, the code for `AnsiString`. When an attempt is made to assign an `AnsiString` to an `OleVariant`, the `AnsiString` will be automatically converted to an OLE `BSTR` and stored in the variant as a `varOleStr`.

Currency

Delphi 2.0 introduced a new type called `Currency`, which is ideal for financial calculations. Unlike floating-point numbers, which allow the decimal point to "float" within a number, `Currency` is a fixed-point decimal type that's hard-coded to a precision of 15 digits before the decimal and four digits after the decimal. As such, it's not susceptible to round-off errors as are floating-point types. When porting your Delphi 1.0 projects, it's a good idea to use this type in place of `Single`, `Real`, `Double`, and `Extended` where money is involved.

User-Defined Types

Integers, strings, and floating-point numbers often are not enough to adequately represent variables in the real-world problems that programmers must try to solve. In cases like these, you must create your own types to better represent variables in the current problem. In Pascal, these user-defined types usually come in the form of records or objects; you declare these types using the `Type` keyword.

Arrays

Object Pascal enables you to create arrays of any type of variable (except files). For example, a variable declared as an array of eight integers reads like this:

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

This statement is equivalent to the following C declaration:

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

It's also equivalent to this Visual Basic statement:

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

Object Pascal arrays have a special property that differentiate them from other languages: They don't have to begin at a certain number. You can therefore declare a three-element array that starts at 28, as in the following example:

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

Because Object Pascal arrays aren't guaranteed to begin at 0 or 1, you must use some care when iterating over array elements in a `for` loop. The compiler provides built-in functions called `High()` and `Low()`, which return the lower and upper bounds of an array variable or

TIP

type, respectively. Your code will be less error prone and easier to maintain if you use these functions to control your `for` loop, as shown here:

```
var
  A: array[28..30] of Integer;
  i: Integer;
begin
  for i := Low(A) to High(A) do  // don't hard-code for loop!
    A[i] := i;
end;
```

Always begin character arrays at 0. Zero-based character arrays can be passed to functions that require `PChar`-type variables. This is a special-case allowance that the compiler provides.

To specify multiple dimensions, use a comma-delimited list of bounds:

```
var
  // Two-dimensional array of Integer:
  A: array[1..2, 1..2] of Integer;
```

To access a multidimensional array, use commas to separate each dimension within one set of brackets:

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

Dynamic Arrays

Dynamic arrays are dynamically allocated arrays in which the dimensions are not known at compile time. To declare a dynamic array, just declare an array without including the dimensions, like this:

```
var
  // dynamic array of string:
  SA: array of string;
```

NOTE

Before you can use a dynamic array, you must use the `SetLength()` procedure to allocate memory for the array:

```
begin
  // allocate room for 33 elements:
  SetLength(SA, 33);
```

Once memory has been allocated, you can access the elements of the dynamic array just like a normal array:

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

Dynamic arrays are always zero-based.

Dynamic arrays are lifetime managed, so there's no need to free them when you're through using them because they'll be released when they leave scope. However, there may come a time when you wish remove the dynamic array from memory before it leaves scope (if it uses a lot of memory, for example) To do this, you need only assign the dynamic array to `nil`:

```
SA := nil;  // releases SA
```

Dynamic arrays are manipulated using reference semantics similar to `AnsiString` types rather than value semantics like a normal array. A quick test: What is the value of `A1[0]` at the end of the following code fragment?

```
var
  A1, A2: array of Integer;
begin
  SetLength(A1, 4);
  A2 := A1;
  A1[0] := 1;
  A2[0] := 26;
```

The correct answer is 26. The reason is because the assignment `A2 := A1` does not create a new array but instead provides `A2` with a reference to the same array as `A1`. Therefore, any modifications to `A2` will also affect `A1`. If you wish instead to make a complete copy of `A1` in `A2`, use the `Copy()` standard procedure:

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

After this line of code is executes, `A2` and `A1` will be two separate arrays initially containing the same data. Changes to one will not affect the other. You can optionally specify the starting element and number of elements to be copied as parameters to `Copy()`, as shown here:

```
// copy 2 elements, starting at element one:
A2 := Copy(A1, 1, 2);
```

Dynamic arrays can also be multidimensional. To specify multiple dimensions, add an additional `array of` to the declaration for each dimension:

```
var
  // two-dimensional dynamic array of Integer:
  IA: array of array of Integer;
```

To allocate memory for a multidimensional dynamic array, pass the sizes of the other dimensions as additional parameters to `SetLength()`:

```
begin
  // IA will be a 5 x 5 array of Integer
  SetLength(IA, 5, 5);
```

You access multidimensional dynamic arrays the same way you do normal multidimensional arrays; each element is separated by a comma with a single set of brackets:

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

Records

A user-defined structure is referred to as a `record` in Object Pascal, and it's the equivalent of C's `struct` or Visual Basic's `Type`. As an example, here's a record definition in Pascal as well as equivalent definitions in C and Visual Basic:

```
{ 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
```

When working with a record, you use the dot symbol to access its fields. Here's an example:

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

```
end;
```

Object Pascal also supports *variant records*, which allow different pieces of data to overlay the same portion of memory in the record. Not to be confused with the `Variant` data type, variant records allow each overlapping data field to be accessed independently. If your background is

NOTE

C/C++, you'll recognize a variant record as being the same concept as a `union` within C `struct`. The following code shows a variant record in which a `Double`, `Integer`, and `char` all

occupy the same memory space:

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

The rules of Object Pascal state that the variant portion of a record cannot be of any lifetime-managed type.

Here's the C++ equivalent of the preceding type declaration:

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

Sets

Sets are a uniquely Pascal type that have no equivalent in Visual Basic, C, or C++ (although Borland C++Builder does implement a template class called `Set`, which emulates the behavior of a Pascal set). Sets provide a very efficient means of representing a collection of ordinal, character, or enumerated values. You can declare a new set type using the keywords `set of` followed by an ordinal type or subrange of possible set values. Here's an example:

```
type
  TCharSet = set of char;      // possible members: #0 - #255
  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday);
  TEnumSet = set of TEnum;  // can contain any combination of TEnum members
  TSubrangeSet = set of 1..10; // possible members: 1 - 10
  TAlphaSet = set of 'A'..'z'; // possible members: 'A' - 'z'
```

Note that a set can only contain up to 256 elements. Additionally, only ordinal types may follow the `set of` keywords. Therefore, the following declarations are illegal:

```
type
  TIntSet = set of Integer;  // Invalid: too many elements
  TStrSet = set of string;   // Invalid: not an ordinal type
```

Sets store their elements internally as individual bits. This makes them very efficient in terms of speed and memory usage. Sets with fewer than 32 elements in the base type can be stored and operated upon in CPU registers, for even greater efficiency. Sets with 32 or more elements (such as a set of `char`–255 elements) are stored in memory. To get the maximum performance benefit from sets, keep the number of elements in the set's base type under 32.

Using Sets

Use square brackets when referencing set elements. The following code demonstrates how to declare set-type variables and assign them values:

```
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;
```

Set Operators

Object Pascal provides several operators for use in manipulating sets. You can use these operators to determine set membership, union, difference, and intersection.

Membership

Use the `in` operator to determine whether a given element is contained in a particular set. For example, the following code would be used to determine whether the `CharSet` set mentioned earlier contains the letter `'S'`:

```
if 'S' in CharSet then
```

TIP

```
  // do something;
```

The following code determines whether `EnumSet` lacks the member `Monday`:

```
if not (Monday in EnumSet) then
```

```
  // do something;
```

Union and Difference

Use the `+` and `-` operators or the `Include()` and `Exclude()` procedures to add and remove elements to and from a set variable:

```
Include(CharSet, 'a');           // add 'a' to set
CharSet := CharSet + ['b'];      // add 'b' to set
Exclude(CharSet, 'x');           // remove 'z' from set
CharSet := CharSet - ['y', 'z']; // remove 'y' and 'z' from set
```

When possible, use `Include()` and `Exclude()` to add and remove a single element to and from a set rather than the `+` and `-` operators. Both `Include()` and `Exclude()` constitute only one machine instruction each, whereas the `+` and `-` operators require 13 + 6*n* (where *n* is the size in bits of the set) instructions.

Intersection

Use the `*` operator to calculate the intersection of two sets. The result of the expression `Set1 * Set2` is a set containing all the members that `Set1` and `Set2` have in common. For example, the following code could be used as an efficient means for determining whether a given set contains multiple elements:

```
if ['a', 'b', 'c'] * CharSet = ['a', 'b', 'c'] then
  // do something
```

Objects

Think of objects as records that also contain functions and procedures. Delphi's object model is discussed in much greater detail later in the "Using Delphi Objects" section of this chapter, so this section covers just the basic syntax of Object Pascal objects. An object is defined as follows:

```
Type
  TChildObject = class(TParentObject);
    SomeVar: Integer;
    procedure SomeProc;
  end;
```

Although Delphi objects are not identical to C++ objects, this declaration is roughly equivalent to the following C++ declaration:

```
class TChildObject : public TParentObject
{
  int SomeVar;
```

NOTE

```
  void SomeProc();
};
```

Methods are defined in the same way as normal procedures and functions (which are discussed in the section "Procedures and Functions"), with the addition of the object name and the dot symbol operator:

```
procedure TChildObject.SomeProc;
```

```
begin
  { procedure code goes here }
end;
```

Object Pascal's `.` symbol is similar in functionality to Visual Basic's `.` operator and C++'s `::` operator. You should note that, although all three languages allow usage of classes, only Object Pascal and C++ allow creation of new classes that behave in a fully object-oriented manner, which we'll describe in the section "Object-Oriented Programming."

NOTE

Object Pascal objects are not laid out in memory the same as C++ objects, so it's not possible to use C++ objects directly from Delphi (and vice versa). However, Chapter 13, "Hard-Core Techniques," shows a technique for sharing objects between C++ and

Delphi.

An exception to this is Borland C++Builder's capability of creating classes that map directly to Object Pascal classes using the proprietary `__declspec(delphiclass)` directive. Such objects are likewise incompatible with regular C++ objects.

Pointers

A *pointer* is a variable that contains a memory location. You already saw an example of a pointer in the `PChar` type earlier in this chapter. Pascal's generic pointer type is called, aptly, `Pointer`. A `Pointer` is sometimes called an untyped pointer because it contains only a memory address, and the compiler doesn't maintain any information on the data to which it points. That notion, however, goes against the grain of Pascal's typesafe nature, so pointers in your code will usually be typed pointers.

NOTE

Pointers are a somewhat advanced topic, and you definitely don't need to master them to write a Delphi application. As you become more experienced, pointers will

become another valuable tool for your programmer's toolbox.

Typed pointers are declared by using the `^` (or *pointer)* operator in the `Type` section of your program. Typed pointers help the compiler keep track of exactly what kind of type a particular

NOTE

pointer points to, thus enabling the compiler to keep track of what you're doing (and can do) with a pointer variable. Here are some typical declarations for pointers:

```
Type
  PInt = ^Integer;        // PInt is now a pointer to an Integer
  Foo = record            // A record type
    GobbledyGook: string;
    Snarf: Real;
  end;
  PFoo = ^Foo;            // PFoo is a pointer to a foo type
var
  P: Pointer;             // Untyped pointer
  P2: PFoo;               // Instance of PFoo
```

C programmers will notice the similarity between Object Pascal's `^` operator and C's `*` operator. Pascal's `Pointer` type corresponds to C's `void *` type.

Remember that a pointer variable only stores a memory address. Allocating space for whatever the pointer points to is your job as a programmer. You can allocate space for a pointer by using one of the memory-allocation routines discussed earlier and shown in Table 2.6.

When a pointer doesn't point to anything (its value is zero), its value is said to be `Nil`, and it is often called a *nil* or *null* pointer.

If you want to access the data that a particular pointer points to, follow the pointer variable name with the `^` operator. This method is known as *dereferencing* the pointer. The following code illustrates working with pointers:

```
Program PtrTest;
Type
  MyRec = record
    I: Integer;
    S: string;
    R: Real;
  end;
  PMyRec = ^MyRec;
var
  Rec : PMyRec;
begin
  New(Rec);      // allocate memory for Rec
  Rec^.I := 10;  // Put stuff in Rec. Note the dereference
  Rec^.S := 'And now for something completely different.';
  Rec^.R := 6.384;
  { Rec is now full }
  Dispose(Rec);  // Don't forget to free memory!
end.
```

When to Use New()

Use the `New()` function to allocate memory for a pointer to a structure of a known size. Because the compiler knows how big a particular structure is, a call to `New()` will cause the correct number of bytes to be allocated, thus making it safer and more convenient to use than `GetMem()` or `AllocMem()`. Never allocate `Pointer` or `PChar` variables by using the `New()` function because the compiler cannot guess how many bytes you need for this allocation. Remember to use `Dispose()` to free any memory you allocate using the `New()` function.

You'll typically use `GetMem()` or `AllocMem()` to allocate memory for structures for which the compiler cannot know the size. The compiler cannot tell ahead of time how much memory you want to allocate for `PChar` or `Pointer` types, for example, because of their variable-length nature. Be careful not to try to manipulate more data than you have allocated with these functions, however, because this is one of the classic causes of an Access Violation error. You should use `FreeMem()` to clean up any memory you allocate with `GetMem()` or `AllocMem()`. `AllocMem()`, by the way, is a bit safer than `GetMem()` because `AllocMem()` always initializes the memory it allocates to zero.

One aspect of Object Pascal that may give C programmers some headaches is the strict type checking performed on pointer types. For example, the variables `a` and `b` in the following example are not type compatible:

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

By contrast, the variables `a` and `b` in the equivalent declaration in C are type compatible:

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

Object Pascal creates a unique type for each pointer-to-type declaration, so you must create a named type if you want to assign values from `a` to `b`, as shown here:

```
type
  PtrInteger = ^Integer;  // create named type
var
  a, b: PtrInteger;       // now a and b are compatible
```

Type Aliases

Object Pascal has the ability to create new names, or *aliases*, for types that are already defined. For example, if you want to create a new name for an `Integer` called `MyReallyNiftyInteger`, you could do so using the following code:

```
type
  MyReallyNiftyInteger = Integer;
```

The newly defined type alias is compatible in all ways with type for which it's an alias. Meaning, in this case, that you could use `MyReallyNiftyInteger` anywhere where you could use `Integer`.

It's possible, however, to define *strongly typed* aliases that are considered new, unique types by the compiler. To do this, use the `type` reserved word in the following manner:

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

Using this syntax, the `MyOtherNeatInteger` type will be converted to an `Integer` when necessary for purposes of assignment, but `MyOtherNeatInteger` will not be compatible with `Integer` when used in `var` and `out` parameters. Therefore, the following code is syntactically correct:

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

On the other hand, the following code will not compile:

```
procedure Goon(var Value: Integer);
begin
  // some code
end;

var
  M: MyOtherNeatInteger;
begin
  M := 29;
  Goon(M);  // Error: M is not var compatible with Integer
```

In addition to these compiler-enforced type compatibility issues, the compiler also generates runtime type information for strongly typed aliases. This enables you to create unique property editors for simple types, as you'll learn in Chapter 22, "Advanced Component Techniques."

Typecasting and Type Conversion

Typecasting is a technique by which you can force the compiler to view a variable of one type as another type. Because of Pascal's strongly typed nature, you'll find that the compiler is very picky about types matching up in the formal and actual parameters of a function call. Hence,

NOTE

you occasionally will be required to cast a variable of one type to a variable of another type to make the compiler happy. Suppose, for example, you need to assign the value of a character to a `byte` variable:

```
var
```

```
  c: char;
  b: byte;
begin
```

```
  c := 's';
  b := c;   // compiler complains on this line
end.
```

In the following syntax, a typecast is required to convert `c` into a `byte`. In effect, a typecast tells the compiler that you really know what you're doing and want to convert one type to another:

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := byte(c);   // compiler happy as a clam on this line
end.
```

You can typecast a variable of one type to another type only if the data size of the two variables is the same. For example, you cannot typecast a `Double` as an `Integer`. To convert a floating-point type to an integer, use the `Trunc()` or `Round()` functions. To convert an integer into a floating-point value, use the assignment operator: `FloatVar := IntVar`.

Object Pascal also supports a special variety of typecasting between objects using the `as` operator, which is described later in the "Runtime Type Information" section of this chapter.

String Resources

Delphi 3 introduced the ability to place string resources directly into Object Pascal source code using the `resourcestring` clause. String resources are literal strings (usually those displayed to the user) that are physically located in a resource attached to the application or library rather than embedded in the source code. Your source code references the string resources in place of string literals. By separating strings from source code, your application can be more easily translated by added string resources in a different language. String resources are declared in the form of *identifier* = *string literal* in the `resourcestring` clause, as shown here:

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

Syntactically, resource strings can be used in your source code in a manner identical to string constants:

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

```
var
  String1: string;

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

NOTE

```
   .
   .
   .
end;
```

Testing Conditions

This section compares `if` and `case` constructs in Pascal to similar constructs in C and Visual Basic. We assume you've used these types of programmatic constructs before, so we don't spend time explaining them to you.

The if Statement

An `if` statement enables you to determine whether certain conditions are met before executing a particular block of code. As an example, here's an `if` statement in Pascal, followed by equivalent definitions in C and Visual Basic:

```
{ Pascal }
if x = 4 then y := x;

/* C */
if (x == 4) y = x;

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

If you have an `if` statement that makes multiple comparisons, make sure you enclose each set of comparisons in parentheses for code clarity. Do this:

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

However, don't do this (it causes the compiler displeasure):

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

Use the `begin` and `end` keywords in Pascal almost as you would use `{` and `}` in C and C++. For example, use the following construct if you want to execute multiple lines of text when a given condition is true:

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

NOTE

You can combine multiple conditions using the `if..else` construct:

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

Using case Statements

The `case` statement in Pascal works in much the same way as a `switch` statement in C and C++. A `case` statement provides a means for choosing one condition among many possibilities without a huge `if..else if..else if` construct. Here's an example of Pascal's `case` statement:

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

The selector type of a `case` statement must be an ordinal type. It's illegal to use nonordinal types, such as strings, as `case` selectors.

Here's the C `switch` statement equivalent to the preceding example:

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

```
          DoSomethingElse; break
  case 303: DoAnotherThing; break;
  default: DoTheDefault;
}
```

Loops

A *loop* is a construct that enables you to repeatedly perform some type of action. Pascal's loop constructs are very similar to what you should be familiar with from your experience with other languages, so this chapter doesn't spend any time teaching you about loops. This section describes the various loop constructs you can use in Pascal.

CAUTION

The for Loop

A `for` loop is ideal when you need to repeat an action a predetermined number of times.

Here's an example, albeit not a very useful one, of a `for` loop that adds the loop index to a variable 10 times:

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

The C equivalent of the preceding example is as follows:

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

Here's the Visual Basic equivalent of the same concept:

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

A caveat to those familiar with Delphi 1: Assignments to the loop control variable are no longer allowed due to the way the loop is optimized and managed by the 32-bit compiler.

The while Loop

Use a `while` loop construct when you want some part of your code to repeat itself while some condition is true. A `while` loop's conditions are tested before the loop is executed, and a classic example for the use of a `while` loop is to repeatedly perform some action on a file as long as the end of the file is not encountered. Here's an example that demonstrates a loop that reads one line at a time from a file and writes it to the screen:

```
Program FileIt;

{$APPTYPE CONSOLE}

var
  f: TextFile;  // a text file
  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's `while` loop works basically the same as C's `while` loop or Visual Basic's `Do While` loop.

repeat..until

The `repeat..until` loop addresses the same type of problem as a `while` loop but from a different angle. It repeats a given block of code until a certain condition becomes `True`. Unlike a `while` loop, the loop code is always executed at least once because the condition is tested at the end of the loop. Pascal's `repeat..until` is roughly equivalent to C's `do..while` loop.

For example, the following code snippet repeats a statement that increments a counter until the value of the counter becomes greater than 100:

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

The Break() Procedure

Calling `Break()` from inside a `while`, `for`, or `repeat` loop causes the flow of your program to skip immediately to the end of the currently executing loop. This method is useful when you need to leave the loop immediately because of some circumstance that may arise within the loop. Pascal's `Break()` procedure is analogous to C's `Break` and Visual Basic's `Exit` statement. The following loop uses `Break()` to terminate the loop after five iterations:

```
var
  i: Integer;
begin
  for i := 1 to 1000000 do
  begin
    MessageBeep(0);          // make the computer beep
    if i = 5 then Break;
  end;
end;
```

The Continue() Procedure

Call `Continue()` inside a loop when you want to skip over a portion of code and the flow of control to continue with the next iteration of the loop. Note in the following example that the code after `Continue()` is not executed in the first iteration of the loop:

```
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;
```

Procedures and Functions

As a programmer, you should already be familiar with the basics of procedures and functions. A *procedure* is a discrete program part that performs some particular task when it's called and then returns to the calling part of your code. A function works the same except that a function returns a value after its exit to the calling part of the program.

If you're familiar with C or C++, consider that a Pascal procedure is equivalent to a C or C++ function that returns `void`, whereas a function corresponds to a C or C++ function that has a return value.

Listing 2.1 demonstrates a short Pascal program with a procedure and a function.

LISTING 2.1 An Example of Functions and Procedures

```
Program FuncProc;

{$APPTYPE CONSOLE}
```

NOTE

```
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.
```

The local variable `Result` in the `IsPositive()` function deserves special attention. Every Object Pascal function has an implicit local variable called `Result` that contains the return value of the function. Note that unlike C and C++, the function doesn't terminate as soon as a value is assigned to `Result`.

You also can return a value from a function by assigning the name of a function to a value inside the function's code. This is standard Pascal syntax and a holdover from previous versions of Borland Pascal. If you choose to use the function name within the body, be careful to note that there is a huge difference between using the function name on the left side of an assignment operator and using it somewhere else in your code. If you use it on the left, you are assigning the function return value. If you use it somewhere else in your code, you are calling the function recursively!

Note that the implicit `Result` variable is not allowed when the compiler's Extended Syntax option is disabled in the Project, Options, Compiler dialog box or when you're using the `{$X-}` directive.

Passing Parameters

Pascal enables you to pass parameters by value or by reference to functions and procedures. The parameters you pass can be of any base or user-defined type or an open array (open arrays are discussed later in this chapter). Parameters also can be constant if their values will not change in the procedure or function.

Value Parameters

Value parameters are the default mode of parameter passing. When a parameter is passed by value, it means that a local copy of that variable is created, and the function or procedure operates on the copy. Consider the following example:

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

When you call a procedure in this way, a copy of string `s` will be made, and `Foo()` will operate on the local copy of `s`. This means that you can choose the value of `s` without having any effect on the variable passed into `Foo()`.

Reference Parameters

Pascal enables you to pass variables to functions and procedures by reference; parameters passed by reference are also called *variable parameters*. Passing by reference means that the function or procedure receiving the variable can modify the value of that variable. To pass a variable by reference, use the keyword `var` in the procedure's or function's parameter list:

```
procedure ChangeMe(var x: longint);
begin
  x := 2;  { x is now changed in the calling procedure }
end;
```

Instead of making a copy of `x`, the `var` keyword causes the address of the parameter to be copied so that its value can be directly modified.

Using `var` parameters is equivalent to passing variables by reference in C++ using the `&` operator. Like C++'s `&` operator, the `var` keyword causes the address of the variable to be passed to the function or procedure rather than the value of the variable.

Constant Parameters

If you don't want the value of a parameter passed into a function to change, you can declare it with the `const` keyword. The `const` keyword not only prevents you from modifying the value of the parameters but it also generates more optimal code for strings and records passed into the procedure or function. Here's an example of a procedure declaration that receives a constant string parameter:

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

Open Array Parameters

Open array parameters provide you with the capability for passing a variable number of arguments to functions and procedures. You can either pass open arrays of some homogenous type or constant arrays of differing types. The following code declares a function that accepts an open array of integers:

```
function AddEmUp(A: array of Integer): Integer;
```

You may pass variables, constants, or constant expressions to open array functions and procedures. The following code demonstrates this by calling `AddEmUp()` and passing a variety of different elements:

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

In order to work with an open array inside the function or procedure, you can use the `High()`, `Low()`, and `SizeOf()` functions in order to obtain information about the array. To illustrate this, the following code shows an implementation of the `AddEmUp()` function that returns the sum of all the numbers passed in `A`:

```
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 also supports an `array of const`, which allows you to pass heterogeneous data types in an array to a function or procedure. The syntax for defining a function or procedure that accepts an `array of const` is as follows:

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

You could call the preceding function with the following syntax:

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

The compiler implicitly converts all parameters to type `TVarRec` when they are passed to the function or procedure accepting the `array of const`. `TVarRec` is defined in the `System` unit as follows:

```
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;
```

The `VType` field indicates what type of data the `TVarRec` contains. This field can have any one of the following values:

```
const
  { TVarRec.VType values }
  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;
```

As you might guess, because `array of const` in the code allows you to pass parameters regardless of their type, they can be difficult to work with on the receiving end. As an example of how to work with `array of const`, the following implementation for `WhatHaveIGot()` iterates through the array and shows a message to the user indicating what type of data was passed in which index:

```
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;
```

Scope

Scope refers to some part of your program in which a given function or variable is known to the compiler. A global constant is in scope at all points in your program, for example, whereas a variable local to some procedure only has scope within that procedure. Consider Listing 2.2.

LISTING 2.2 An Illustration of Scope

```
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`, and `R` have global scope—their values are known to the compiler at all points within the program. Procedure `SomeProc()` has two variables in which the scope is local to that procedure: `R` and `LocalReal`. If you try to access `LocalReal` outside of `SomeProc()`, the compiler displays an unknown identifier error. If you access `R` within `SomeProc()`, you'll be referring to the local version, but if you access `R` outside that procedure, you'll be referring to the global version.

Units

Units are the individual source code modules that make up a Pascal program. A unit is a place for you to group functions and procedures that can be called from your main program. To be a unit, a source module must consist of at least three parts:

- *A* `unit` *statement.* Every unit must have as its first line a statement saying that it's a unit and identifying the unit name. The name of the unit always must match the filename. For example, if you have a file named `FooBar`, the statement would be

```
unit FooBar;
```

- *The* `interface` *part.* After the `unit` statement, a unit's next functional line of code

NOTE

should be the `interface` statement. Everything following this statement, up to the `implementation` statement, is information that can be shared with your program and with other units. The `interface` part of a unit is where you declare the types, constants, variables, procedures, and functions that you want to make available to your main program and to other units. Only declarations—never procedure bodies—can appear in the interface. The `interface` statement should be one word on one line:

```
interface
```

- *The* `implementation` *part.* This follows the `interface` part of the unit. Although the `implementation` part of the unit contains primarily procedures and functions, it's also where you declare any types, constants, and variables that you do not want to make available outside of this unit. The `implementation` part is where you define any functions or procedures that you declared in the `interface` part. The `implementation` statement should be one word on one line:

```
implementation
```

Optionally, a unit can also include two other parts:

- *An* `initialization` *part.* This portion of the unit, which is located near the end of the file, contains any initialization code for the unit. This code will be executed before the main program begins execution, and it executes only once.
- *A* `finalization` *part.* This portion of the unit, which is located in between the `initialization` and `end.` of the unit, contains any cleanup code that executes when the program terminates. The `finalization` section was introduced to the language in Delphi 2.0. In Delphi 1.0, unit finalization was accomplished by adding a new exit procedure using the `AddExitProc()` function. If you're porting an application from Delphi 1.0, you should move your exit procedures into the finalization part of your units.

When several units have `initialization/finalization` code, execution of each section proceeds in the order in which the units are encountered by the compiler (the first unit in the program's `uses` clause, then the first unit in that unit's `uses` clause, and so on). Also, it's a bad idea to write initialization and finalization code that relies on such ordering because one small change to the `uses` clause can cause some difficult-to-find bugs!

The uses Clause

The `uses` clause is where you list the units that you want to include in a particular program or unit. For example, if you have a program called FooProg that uses functions and types in two units, `UnitA` and `UnitB`, the proper `uses` declaration is as follows:

```
Program FooProg;


uses UnitA, UnitB;
```

Units can have two `uses` clauses: one in the `interface` section and one in the `implementation` section.

Here's code for a sample unit:

```
Unit FooBar;

interface

uses BarFoo;

  { public declarations here }

implementation

uses BarFly;

  { private declarations here }

initialization
  { unit initialization here }
finalization
  { unit clean-up here }
end.
```

Circular Unit References

Occasionally, you'll have a situation where `UnitA` uses `UnitB` and `UnitB` uses `UnitA`. This is called a *circular unit reference*. The occurrence of a circular unit reference is often an indication of a design flaw in your application; you should avoid structuring your program with a circular reference. The optimal solution is often to move a piece of data that both `UnitA` and `UnitB` need to use out to a third unit. However, as with most things, sometimes you just can't avoid the circular unit reference. In such a case, move one of the `uses` clauses to the `implementation` part of your unit and leave the other one in the `interface` part. This usually solves the problem.

Packages

Delphi *packages* enable you to place portions of your application into separate modules, which can be shared across multiple applications. If you already have an existing investment in Delphi 1 or 2 code, you'll appreciate that you can take advantage of packages without any changes to your existing source code.

Think of a package as a collection of units stored in a separate DLL-like module (a Borland Package Library, or *BPL file).* Your application can then link with these "packaged" units at runtime rather than compile/link time. Because the code for these units resides in the BPL file rather than in your EXE or DLL, the size of your EXE or DLL can become very small. Four types of packages are available for you to create and use:

- *Runtime package.* This type of package contains units required at runtime by your application. When compiled to depend on a particular runtime package, your application will not run in the absence of that package. Delphi's `VCL50.BPL` is an example of this type of package.
- *Design package.* This type of package contains elements necessary for application design such as components, property and component editors, and experts. It can be installed into Delphi's component library using the Component, Install Package menu item. Delphi's `DCL*.BPL` packages are examples of this type of package. This type of package is described in more detail in Chapter 21, "Writing Delphi Custom Components."
- *Runtime and design package.* This package serves both of the purposes listed in the first two items. Creating this type of package makes application development and distribution a bit simpler, but this type of package is less efficient because it must carry the baggage of design support even in your distributed applications.
- *Neither runtime nor design package.* This rare breed of package is intended to be used only by other packages and is not intended to be referenced directly by an application or used in the design environment.

Using Delphi Packages

Package-enabling your Delphi applications is easy. Simply check the Build with Runtime Packages check box in the Project, Options, Packages dialog box. The next time you build your application after selecting this option, your application will be linked dynamically to runtime packages rather than having units linked statically into your EXE or DLL. The result will be a much more svelte application (although bear in mind that you'll have to deploy the necessary packages with your application).

Package Syntax

Packages are most commonly created using the Package Editor, which you invoke by choosing the File, New, Package menu item. This editor generates a *Delphi Package Source* (DPK) file, which will be compiled into a package. The syntax for this DPK file is quite simple, and it uses the following format:

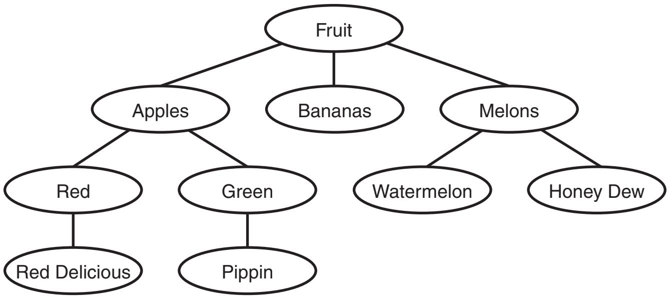
```
package PackageName

requires Package1, Package2, ...;

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

  ...;
end.
```

Packages listed in the `requires` clause are required in order for this package to load. Typically, packages containing units used by units listed in the `contains` clause are listed here. Units listed in the `contains` clause will be compiled into this package. Note that units listed here must not also be listed in the `contains` clause of any of the packages listed in the `requires` clause. Note also that any units used by units in the `contains` clause will be implicitly pulled into this package (unless they're contained in a required package).



Object-Oriented Programming

Volumes have been written on the subject of *object-oriented programming* (OOP). Often, OOP seems more like a religion than a programming methodology, spawning arguments about its merits (or lack thereof) passionate and spirited enough to make the Crusades look like a slight disagreement. We're not orthodox OOPists, and we're not going to get involved in the relative merits of OOP; we just want to give you the lowdown on a fundamental principle on which Delphi's Object Pascal Language is based.

OOP is a programming paradigm that uses discrete objects—containing both data and code—as application building blocks. Although the OOP paradigm doesn't necessarily lend itself to easier-to-write code, the result of using OOP traditionally has been easy-to-maintain code. Having objects' data and code together simplifies the process of hunting down bugs, fixing them with minimal effect on other objects, and improving your program one part at a time. Traditionally, an OOP language contains implementations of at least three OOP concepts:

- *Encapsulation.* Deals with combining related data fields and hiding the implementation details. The advantages of encapsulation include modularity and isolation of code from other code.

- *Inheritance.* The capability to create new objects that maintain the properties and behavior of ancestor objects. This concept enables you to create object hierarchies such as VCL—first creating generic objects and then creating more specific descendants of those objects that have more narrow functionality.

  The advantage of inheritance is the sharing of common code. Figure 2.4 presents an example of inheritance—how one root object, fruit, is the ancestor object of all fruits, including the melon. The melon is ancestor of all melons, including the watermelon. You get the picture.

FIGURE 2.4
An illustration of inheritance.

NOTE

- *Polymorphism.* Literally, *polymorphism* means "many shapes." Calls to methods of an object variable will call code appropriate to whatever instance is actually in the variable.

A Note on Multiple Inheritance

Object Pascal does not support multiple inheritance of objects as C++ does. *Multiple inheritance* is the concept of a given object being derived from two separate objects, creating an object that contains all the code and data of the two parent objects.

To expand on the analogy presented in Figure 2.4, multiple inheritance enables you to create a candy apple object by creating a new object that inherits from the apple class and some other class called "candy." Although this functionality seems useful, it often introduces more problems and inefficiencies into your code than it solves.

Object Pascal provides two approaches to solving this problem. The first solution is to make one class *contain* the other class. You'll see this solution throughout Delphi's VCL. To build upon the candy apple analogy, you would make the candy object a member of the apple object. The second solution is to use *interfaces* (you'll learn more about interfaces in the section "Interfaces"). Using interfaces, you could essentially have one object that supports both a candy and an apple interface.

You should understand the following three terms before you continue to explore the concept of objects:

- *Field.* Also called *field definitions* or *instance variables*, fields are data variables contained within objects. A field in an object is just like a field in a Pascal record. In C++, fields sometimes are referred to as *data members*.
- *Method.* The name for procedures and functions belonging to an object. Methods are called *member functions* in C++.
- *Property.* An entity that acts as an accessor to the data and code contained within an object. Properties insulate the end user from the implementation details of an object.

It's generally considered bad OOP style to access an object's fields directly. This is because the implementation details of the object may change. Instead, use *accessor properties,* which allow a standard object interface without becoming embroiled in the details of how the objects are implemented. Properties are explained in the "Properties" section later in this chapter.

Object-Based Versus Object-Oriented Programming

In some tools, you manipulate entities (objects), but you cannot create your own objects. ActiveX (formerly OCX) controls in Visual Basic are a good example of this. Although you can use an ActiveX control in your applications, you cannot create one, and you cannot inherit one ActiveX control from another in Visual Basic. Environments such as these often are called *object-based environments*.

Delphi is a fully object-oriented environment. This means that you can create new objects in Delphi either from scratch or based on existing components. This includes all Delphi objects, be they visual, nonvisual, or even design-time forms.

Using Delphi Objects

As mentioned earlier, objects (also called *classes)* are entities that can contain both data and code. Delphi objects also provide you with all the power of object-oriented programming in offering full support of inheritance, encapsulation, and polymorphism.

Declaration and Instantiation

Of course, before using an object, you must have declared an object using the `class` keyword.

NOTE

As described earlier in this chapter, objects are declared in the `type` section of a unit or program:

```
type
```

```
  TFooObject = class;
```

In addition to an object type, you usually also will have a variable of that class type, or *instance*, declared in the `var` section:

```
var
  FooObject: TFooObject;
```

You create an instance of an object in Object Pascal by calling one of its *constructors*. A constructor is responsible for creating an instance of your object and allocating any memory or initializing any fields necessary so that the object is in a usable state upon exiting the constructor. Object Pascal objects always have at least one constructor called `Create()`—although it's

CAUTION

possible for an object to have more than one constructor. Depending on the type of object, `Create()` can take different numbers of parameters. This chapter focuses on the simple case where `Create()` takes no parameters.

Unlike C++, object constructors in Object Pascal are not called automatically, and it's incumbent on the programmer to call the object constructor. The syntax for calling a constructor is as follows:

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

Notice that the syntax for a constructor call is a bit unique. You're referencing the `Create()` method of the object by the type rather than the instance, as you would with other methods. This may seem odd at first, but it does make sense. `FooObject`, a variable, is undefined at the time of the call, but the code for `TFooObject`, a type, is static in memory. A static call to its

`Create()` method is therefore totally valid.

The act of calling a constructor to create an instance of an object is often called *instantiation*.

When an object instance is created using the constructor, the compiler will ensure that every field in your object is initialized. You can safely assume that all numbers will be initialized to `0`, all pointers to `Nil`, and all strings will be empty.

Destruction

When you're finished using an object, you should deallocate the instance by calling its `Free()` method. The `Free()` method first checks to ensure that the object instance is not `Nil`; then it calls the object's *destructor* method, `Destroy()`. The destructor, of course, does the opposite of the constructor; it deallocates any allocated memory and performs any other housekeeping required in order for the object to be properly removed from memory. The syntax is simple:

```
FooObject.Free;
```

Unlike the call to `Create()`, the object instance is used in the call to the `Free()` method. Remember never to call `Destroy()` directly but instead to call the safer `Free()` method.

In C++, the destructor of an object declared statically is called automatically when your object leaves scope, but you must call the destructor for any dynamically allocated objects. The rule is the same in Object Pascal, except that all objects are implicitly dynamic in Object Pascal, so you must follow the rule of thumb that anything you create, you must free. There are a couple of important exceptions to this rule, however. The first is when your object is owned by other objects (as described in Chapter 20, "Key Elements of the Visual Component Library"), it will be freed for you. The second is reference counted objects (such as those descending from `TInterfacedObject` or `TComObject`), which are destroyed when the last reference is released.

You might be asking yourself how all these methods got into your little object. You certainly didn't declare them yourself, right? Right. The methods just discussed actually come from the Object Pascal's base `TObject` object. In Object Pascal, all objects are always descendants of `TObject` regardless of whether they're declared as such. Therefore, the declaration

```
Type TFoo = Class;
```

is equivalent to the declaration

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

Methods

Methods are procedures and functions belonging to a given object. Methods are those things that give an object behavior rather than just data. Two important methods of the objects you create are the constructor and the destructor methods, which we just covered. You can also create custom methods in your objects to perform a variety of tasks.

Creating a method is a two-step process. You first must declare the method in the object type declaration, and then you must define the method in the code. The following code demonstrates the process of declaring and defining a method:

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

Note that when defining the method body, you have to use the fully qualified name, as you did when defining the `DoTheHustle` method. It's important also to note that the object's `Dance` field can be accessed directly from within the method.

Method Types

Object methods can be declared as `static`, `virtual`, `dynamic`, or `message`. Consider the following example object:

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

Static Methods

`IAmAStatic` is a static method. The *static* method is the default method type, and it works similarly to a regular procedure or function call. The compiler knows the address of these methods, and so, when you call a static method, it's able to link that information into the executable statically. Static methods execute the fastest; however, they do not have the ability to be overridden to provide *polymorphism*.

Virtual Methods

`IAmAVirtual` is a virtual method. *Virtual* methods are called in the same way as static methods, but because virtual methods can be overridden, the compiler does not know the address of a particular virtual function when you call it in your code. The compiler, therefore, builds a *Virtual Method Table* (VMT) that provides a means to look up function addresses at runtime. All virtual method calls are dispatched at runtime through the VMT. An object's VMT contains all its ancestor's virtual methods as well as the ones it declares; therefore, virtual methods use more memory than dynamic methods, although they execute faster.

Dynamic Methods

`IAmADynamic` is a dynamic method. *Dynamic* methods are basically virtual methods with a different dispatching system. The compiler assigns a unique number to each dynamic method and uses those numbers, along with method addresses, to build a *Dynamic Method Table* (DMT). Unlike the VMT, an object's DMT contains only the dynamic methods that it declares, and that method relies on its ancestor's DMTs for the rest of its dynamic methods. Because of this, dynamic methods are less memory intensive than virtual methods, but they take longer to call because you may have to propagate through several ancestor DMTs before finding the address of a particular dynamic method.

Message Methods

`IAmAMessage` is a message-handling method. The value after the `message` keyword dictates what message the method will respond to. Message methods are used to create an automatic response to Windows messages, and you generally don't call them directly. Message handling is discussed in detail in Chapter 5, "Understanding Messages."

Overriding Methods

Overriding a method is Object Pascal's implementation of the OOP concept of polymorphism. It enables you to change the behavior of a method from descendant to descendant. Object Pascal methods can be overridden only if they're first declared as `virtual` or `dynamic`. To override a method, just use the `override` directive instead of `virtual` or `dynamic` in your descendant object type. For example, you could override the `IAmAVirtual` and `IAmADynamic` methods as shown here:

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

The `override` directive replaces the original method's entry in the VMT with the new method. If you had re-declared `IAmAVirtual` and `IAmADynamic` with the `virtual` or `dynamic` keyword instead of `override`, you would have created new methods rather than overriding the ancestor methods. Also, if you attempt to override a static method in a descendant type, the static method in the new object completely replaces the method in the ancestor type.

Method Overloading

Like regular procedures and functions, methods can be overloaded so that a class can contain multiple methods of the same name with differing parameter lists. Overloaded methods must be marked with the `overload` directive, although the use of the directive on the first instance of a method name in a class hierarchy is optional. The following code example shows a class containing three overloaded methods:

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

Reintroducing Method Names

Occasionally, you may want to add a method to one of your classes to replace a method of the same name in an ancestor of your class. In this case, you don't want to override the ancestor method but instead obscure and completely supplant the base class method. If you simply add the method and compile, you'll see that the compiler will produce a warning explaining that the new method hides a method of the same name in a base class. To suppress this error, use the `reintroduce` directive on the method in the ancestor class. The following code example demonstrates proper use of the `reintroduce` directive:

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

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

Self

An implicit variable called `Self` is available within all object methods. `Self` is a pointer to the class instance that was used to call the method. `Self` is passed by the compiler as a hidden parameter to all methods.

Properties

It may help to think of properties as special accessor fields that enable you to modify data and execute code contained within your class. For components, properties are those things that show up in the Object Inspector window when published. The following example illustrates a simplified `Object` with a property:

```
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` is an object that contains the following: one field (an integer called `SomeValue`), one method (a procedure called `SetSomeValue`), and one property called `Value`. The sole purpose of the `SetSomeValue` procedure is to set the value of the `SomeValue` field. The `Value` property doesn't actually contain any data. `Value` is an accessor for the `SomeValue` field; when you ask `Value` what number it contains, it reads the value from `SomeValue`. When you attempt to set the value of the `Value` property, `Value` calls `SetSomeValue` to modify the value of `SomeValue`. This is useful for two reasons: First, it allows you to present the users of the class with a simple variable without making them worry about the class's implementation details. Second, you can allow the users to override accessor methods in descendant classes for polymorphic behavior.

Visibility Specifiers

Object Pascal offers you further control over the behavior of your objects by enabling you to declare fields and methods with directives such as `protected`, `private`, `public`, `published`, and `automated`. The syntax for using these keywords is as follows:

```
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;
```

You can place as many fields or methods as you want under each directive. Style dictates that you should indent the specifier the same as you indent the class name. The meanings of these directives follow:

- `private`. These parts of your object are accessible only to code in the same unit as your object's implementation. Use this directive to hide implementation details of your objects from users and to prevent users from directly modifying sensitive members of your object.

- `protected`. Your object's `protected` members can be accessed by descendants of your object. This capability enables you to hide the implementation details of your object from users while still providing maximum flexibility to descendants of your object.
- `public`. These fields and methods are accessible anywhere in your program. Object constructors and destructors always should be `public`.
- `published`. *Runtime Type Information* (RTTI) to be generated for the published portion of your objects enables other parts of your application to get information on your object's published parts. The Object Inspector uses RTTI to build its list of properties.
- `automated`. The `automated` specifier is obsolete but remains for compatibility with Delphi 2. Chapter 23, "COM and ActiveX," has more details on this.

Here, then, is code for the `TMyObject` class that was introduced earlier, with directives added to improve the integrity of the object:

```
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;
```

Now, users of your object will not be able to modify the value of `SomeValue` directly, and they will have to go through the interface provided by the property `Value` to modify the object's data.

"Friend" Classes

The C++ language has a concept of *friend* classes (that is, classes that are allowed access to the private data and functions in other classes). This is accomplished in C++ using the `friend` keyword. Although, strictly speaking, Object Pascal doesn't have a similar keyword, it does allow for similar functionality. All objects declared within the same unit are considered "friends" and are allowed access to the private information located in other objects in that unit.

Inside Objects

All class instances in Object Pascal are actually stored as 32-bit pointers to class instance data located in heap memory. When you access fields, methods, or properties within a class, the compiler automatically performs a little bit of hocus-pocus that generates the code to dereference that pointer for you. Therefore, to the untrained eye, a class appears as a static variable. What this means, however, is that unlike C++, Object Pascal offers no reasonable way to allocate a class from an application's data segment other than from the heap.

TObject: The Mother of All Objects

Because everything descends from `TObject`, every class has some methods that it inherits from `TObject`, and you can make some special assumptions about the capabilities of an object. Every class has the ability, for example, to tell you its name, its type, or even whether it's inherited from a particular class. The beauty of this is that you, as an applications programmer, don't have to care what kind of magic the compiler does to makes this happen. You can just take advantage of the functionality it provides!

`TObject` is a special object because its definition comes from the `System` unit, and the Object Pascal compiler is "aware" of `TObject`. The following code illustrates the definition of the `TObject` class:

```
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;
```

You'll find each of these methods documented in Delphi's online help system.

In particular, note the methods that are preceded by the keyword `class`. Prepending the `class` keyword to a method enables it to be called like a normal procedure or function without actually having an instance of the class of which the method is a member. This is a juicy bit of functionality that was borrowed from C++'s `static` functions. Be careful, though, not to make a class method depend on any instance information; otherwise, you'll get a compiler error.

Interfaces

Perhaps the most significant addition to the Object Pascal language in the recent past is the native support for *interfaces*, which was introduced in Delphi 3. Simply put, an interface defines a set of functions and procedures that can be used to interact with an object. The defin-

TIP

ition of a given interface is known to both the implementer and the client of the interface—acting as a contract of sorts for how an interface will be defined and used. A class can implement

multiple interfaces, providing multiple known "faces" by which a client can control an object.

As its name implies, an interface defines only, well, an interface by which object and clients communicate. This is similar in concept to a C++ `PURE VIRTUAL` class. It's the job of a class that supports an interface to implement each of the interface's functions and procedures.

In this chapter you'll learn about the language elements of interfaces. For information on using interfaces within your applications, see Chapter 23, "COM and ActiveX."

Defining Interfaces

Just as all Delphi classes implicitly descend from `TObject`, all interfaces are implicitly derived from an interface called `IUnknown`. `IUnknown` is defined in the `System` unit as follows:

```
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;
```

As you can see, the syntax for defining an interface is very similar to that of a class. The primary difference is that an interface can optionally be associated with a *globally unique identifier* (GUID), which is unique to the interface. The definition of `IUnknown` comes from the *Component Object Model* (COM) specification provided by Microsoft. This is also described in more detail in Chapter 23, "COM and ActiveX."

Defining a custom interface is straightforward if you understand how to create Delphi classes. The following code defines a new interface called `IFoo`, which implements one method called `F1()`:

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

The Delphi IDE will manufacture new GUIDs for your interfaces when you use the Ctrl+Shift+G key combination.

The following code defines a new interface, `IBar`, which descends from `IFoo`:

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

Implementing Interfaces

The following bit of code demonstrates how to implement `IFoo` and `IBar` in a class called `TFooBar`:

```
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;
```

Note that multiple interfaces can be listed after the ancestor class in the first line of the class declaration in order to implement multiple interfaces. The binding of an interface function to a particular function in the class happens when the compiler matches a method signature in the interface with a matching signature in the class. A compiler error will occur if a class declares that it implements an interface but the class fails to implement one or more of the interface's methods.

If a class implements multiple interfaces that have methods of the same signature, you must alias the same-named methods as shown in the short example following:

```
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)
    // aliased methods
    function IFoo.F1 = FooF1;
    function IBar.F1 = BarF1;
    // interface methods
    function FooF1: Integer;
    function BarF1: Integer;
  end;

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

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

The implements Directive

Delphi 4 introduced the `implements` directive, which enables you to delegate the implementation of interface methods to another class or interface. This technique is sometimes called *implementation by delegation*. `Implements` is used as the last directive on a property of class or interface type like this:

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

The use of `implements` in the preceding code example instructs the compiler to look to the `Foo` property for the methods that implement the `IFoo` interface. The type of the property must be a class that contains `IFoo` methods or an interface of type `IFoo` or a descendant of `IFoo`. You can also provide a comma-delimited list of interfaces following the `implements` directive, in which case the type of the property must contain the methods to implement the multiple interfaces.

The `implements` directive buys you two key advantages in your development: First, it allows you to perform aggregation in a no-hassle manner. Aggregation is a COM concept pertaining to the combination of multiple classes for a single purpose (see Chapter 23, "COM and ActiveX," for more information on aggregation). Second, it allows you to defer the consumption of resources necessary to implement an interface until it's absolutely necessary. For example, say there was an interface whose implementation requires allocation of a 1MB bitmap, but that interface is seldom required by clients. You probably wouldn't want to implement that interface all the time "just in case" because that would be a waste of resources. Using `implements`, you could create the class to implement the interface on demand in the property accessor method.

Using Interfaces

A few important language rules apply when you're using variables of interface types in your applications. The foremost rule to remember is that an interface is a lifetime-managed type. This means it's always initialized to `nil`, it's reference counted, a reference is automatically added when you obtain an interface, and it's automatically released when it leaves scope or is assigned the value `nil`. The following code example illustrates the lifetime management of an interface variable:

```
var
  I: ISomeInterface;
begin
  // I is initialized to nil
  I := FunctionReturningAnInterface;  // ref count of I is incremented
  I.SomeFunc;
  // ref count of I is decremented.  If 0, I is automatically released
end;
```

Another unique rule of interface variables is that an interface is assignment compatible with classes that implement the interface. For example, the following code is legal using the `TFooBar` class defined earlier:

```
procedure Test(FB: TFooBar)
var

  F: IFoo;
begin

  F := FB;  // legal because FB supports IFoo
  .
  .
  .
```

Finally, the `as` typecast operator can be used to `QueryInterface` a given interface variable for another interface (this is explained in greater detail in Chapter 23). This is illustrated here:

```
var
  FB: TFooBar;
  F: IFoo;
  B: IBar;
begin
  FB := TFooBar.Create
  F := FB;  // legal because FB supports IFoo
  B := F as IBar;  // QueryInterface F for IBar
  .
  .
  .
```

If the requested interface is not supported, an exception will be raised.

Structured Exception Handling

Structured exception handling (SEH) is a method of error handling that enables your application to recover gracefully from otherwise fatal error conditions. In Delphi 1, exceptions were implemented in the Object Pascal language, but starting in Delphi 2, exceptions are a part of the Win32 API. What makes Object Pascal exceptions easy to use is that they're just classes that happen to contain information about the location and nature of a particular error. This

NOTE

makes exceptions as easy to implement and use in your applications as any other class.

Delphi contains predefined exceptions for common program-error conditions, such as out of memory, divide by zero, numerical overflow and underflow, and file I/O errors. Delphi also enables you to define your own exception classes as you may see fit in your applications.

Listing 2.3 demonstrates how to use exception handling during file I/O.

LISTING 2.3 File I/O using exception handling

```
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.
```

In Listing 2.3, the inner `try..finally` block is used to ensure that the file is closed regardless of whether any exceptions come down the pike. What this block means in English is "Hey, program, try to execute the statements between the `try` and the `finally`. If you finish them or run into an exception, execute the statements between the `finally` and the `end`. If an exception does occur, move on to the next exception-handling block." This means that the file will be closed and the error can be properly handled no matter what error occurs.

The statements after `finally` in a `try..finally` block execute regardless of whether an exception occurs. Make sure that the code in your `finally` block does not assume that an exception has occurred. Also, because the `finally` statement doesn't stop the migration of an exception, the flow of your program's execution will continue on to the next exception handler.

The outer `try..except` block is used to handle the exceptions as they occur in the program. After the file is closed in the `finally` block, the `except` block puts up a message informing the user that an I/O error occurred.

One of the key advantages that exception handling provides over the traditional method of error handling is the ability to distinctly separate the error-detection code from the error-correction code. This is a good thing primarily because it makes your code easier to read and maintain by enabling you to concentrate on one distinct aspect of the code at a time.

The fact that you cannot trap any specific exception by using the `try..finally` block is significant. When you use a `try..finally` block in your code, it means that you don't care what exceptions might occur. You just want to perform some tasks when they do occur to gracefully get out of a tight spot. The `finally` block is an ideal place to free any resources you've allocated (such as files or Windows resources), because it will always execute in the case of an

CAUTION

error. In many cases, however, you need some type of error handling that's able to respond differently depending on the type of error that occurs. You can trap specific exceptions by using a `try..except` block, which is again illustrated in Listing 2.4.

LISTING 2.4 A `try..except` exception-handling block

```
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.
```

Although you can trap specific exceptions with the `try..except` block, you also can catch other exceptions by adding the catchall `else` clause to this construct. The syntax of the `try..except..else` construct follows:

```
try
  Statements
except
  On ESomeException do Something;
else
  { do some default exception handling }
end;
```

When using the `try..except..else` construct, you should be aware that the `else` part will catch *all* exceptions—even exceptions you might not expect, such as out-of-memory or other runtime-library exceptions. Be careful when using the `else` clause, and use the clause sparingly. You should always reraise the exception when you trap with unqualified exception handlers. This is explained in the section "Reraising an Exception."

You can achieve the same effect as a `try..except..else` construct by not specifying the exception class in a `try..except` block, as shown in this example:

```
try
```

CAUTION

```
  Statements
except
  HandleException  // almost the same as else statement
```

```
end;
```

Exception Classes

Exceptions are merely special instances of objects. These objects are instantiated when an exception occurs and are destroyed when an exception is handled. The base exception object is called `Exception`, and that object is defined as follows:

```
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;
```

The important element of the `Exception` object is the `Message` property, which is a string. `Message` provides more information or explanation on the exception. The information provided by `Message` depends on the type of exception that's raised.

If you define your own exception object, make sure that you derive it from a known exception object such as `Exception` or one of its descendants. The reason for this is so that generic exception handlers will be able to trap your exception.

When you handle a specific type of exception in an `except` block, that handler also will catch any exceptions that are descendants of the specified exception. For example, `EMathError` is the ancestor object for a variety of math-related exceptions, such as `EZeroDivide` and `EOverflow`. You can catch any of these exceptions by setting up a handler for `EMathError`, as shown here:

```
try
  Statements
except
  on EMathError do  // will catch EMathError or any descendant
    HandleException
end;
```

Any exceptions that you do not explicitly handle in your program eventually will flow to, and be handled by, the default handler located within the Delphi runtime library. The default handler will put up a message dialog box informing the user that an exception occurred. Incidentally, Chapter 4, "Application Frameworks and Design Concepts," will show an example of how to override the default exception handling.

When handling an exception, you sometimes need to access the instance of the exception object in order to retrieve more information on the exception, such as that provided by its `Message` property. There are two ways to do this: Use an optional identifier with the `on ESomeException` construct or use the `ExceptObject()` function.

TIP

You can insert an optional identifier in the `on ESomeException` portion of an `except` block and have the identifier map to an instance of the currently raised exception. The syntax for this is to preface the exception type with an identifier and a colon, as follows:

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

In this case, the identifier (`E` in this case) becomes the instance of the currently raised exception. This identifier is always of the same type as the exception it prefaces.

You can also use the `ExceptObject()` function, which returns an instance of the currently raised exception. The drawback to `ExceptObject()`, however, is that it returns a `TObject` that you must then typecast to the exception object of your choice. The following example shows the usage of this function:

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

The `ExceptObject()` function will return `Nil` if there is no active exception.

The syntax for raising an exception is similar to the syntax for creating an object instance. To raise a user-defined exception called `EBadStuff`, for example, you would use this syntax:

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

Flow of Execution

After an exception is raised, the flow of execution of your program propagates up to the next exception handler until the exception instance is finally handled and destroyed. This process is determined by the call stack and therefore works program-wide (not just within one procedure or unit). Listing 2.5 illustrates the flow of execution of a program when an exception is raised. This listing is the main unit of a Delphi application that consists of one form with one button on the form. When the button is clicked, the `Button1Click()` method calls `Proc1()`, which calls `Proc2()`, which in turn calls `Proc3()`. An exception is raised in `Proc3()`, and you can witness the flow of execution propagating through each `try..finally` block until the exception is finally handled inside `Button1Click()`.

When you run this program from the Delphi IDE, you'll be able to see the flow of execution better if you disable the integrated debugger's handling of exceptions by unchecking Tools, Debugger Options, Language Exceptions, Stop on Delphi Exceptions.

LISTING 2.5 Main unit for the exception propagation project

```
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.
```

Reraising an Exception

When you need to perform special handling for a statement inside an existing `try..except` block and still need to allow the exception to flow to the block's outer default handler, you can use a technique called *reraising the exception*. Listing 2.6 demonstrates an example of reraising an exception.

LISTING 2.6 Reraising an exception

```
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;
```

Runtime Type Information

Runtime Type Information (RTTI) is a language feature that gives a Delphi application the capability to retrieve information about its objects at runtime. RTTI is also the key to links between Delphi components and their incorporation into the Delphi IDE, but it isn't just an academic process that occurs in the shadows of the IDE.

Objects, by virtue of being `TObject` descendants, contain a pointer to their RTTI and have several built-in methods that enable you to get some useful information out of the RTTI. The following table lists some of the `TObject` methods that use RTTI to retrieve information about a particular object instance.

Function	*Return Type*	*Returns*
`ClassName()`	`string`	The name of the object's class
`ClassType()`	`TClass`	The object's type
`InheritsFrom()`	`Boolean`	Boolean to indicate whether the class descends from a given class
`ClassParent()`	`TClass`	The object ancestor's type
`InstanceSize()`	`word`	The size, in bytes, of an instance
`ClassInfo()`	`Pointer`	A pointer to the object's in-memory RTTI