# Basic Pascal Syntax

CHAPTER

3

*By Charlie Calvert*

## IN THIS CHAPTER

This chapter is dedicated to some of the more advanced features of the most commonly used Pascal types. Subjects covered include integers, strings, floating-point types, arrays, records, and pointers.

I'm writing for experienced programmers, and the text is not meant to be a primer on Object Pascal for newcomers. I assume that the reader already understands simple types such as integers, floating-point types, and even complex types such as arrays, records, and pointers. I will highlight the unexpected or advanced features of these types as they are implemented in Object Pascal.

Some readers might have last used Pascal many years ago and might no longer have a sure feeling of the language. Many programmers have had experience with ANSI Pascal but not with Object Pascal. Object Pascal is to ANSI Pascal as C++ is to C. If you're expecting the Object Pascal universe to resemble the relatively bland world of ANSI Pascal, then you may be in for some surprises.

The material in this chapter will be necessary for an understanding of the material in Chapter 5, "The Editor and Debugger." In short, I can't discuss the finer points of writing and debugging code in the Object Pascal editor without first covering a few crucial syntactical issues.

**DELPHI NOTE**

Experienced Object Pascal programmers will want to at least skim this chapter because there might be some issues such as variant `records` or dynamic arrays that you want to bone up on. Advanced topics are spread throughout the chapter. For instance, one section that even experienced Delphi users should read is "The `TBcd` type and Floating-Point Accuracy," which covers the new routines for handling the `TBcd` type.

## The Content of This Chapter

Pascal is a strongly typed language. The compiler cares a great deal about your type declarations, and the things you can do with variables are severely restricted by your type choices. In the old days, this was considered a bad thing because it was felt to be restricting. Over the years, however, the general consensus on this topic has changed. All modern languages, such as C++ and Java, have followed in Pascal's footsteps by placing a strong emphasis on typing.

Pascal was invented back in 1968 and was first implemented in 1970. Any language that old is going to have some quirks in it. However, overall, I consider Pascal to be a very good language. If you come to this book with a different prejudice, I ask that you put those ideas aside

while you read this chapter. You might be surprised at the power of this language. In particular, C++ programmers probably will be surprised to find that the language is so flexible, and Java programmers will be surprised to find that it has so many modern features.

Most major Pascal types will be covered in one fashion or another. However, there are two major topics that I will just touch on:

- **Objects**—The Object Pascal syntax for declaring and building objects will be explored in Chapter 4, "Objects and Interfaces," and also throughout nearly all the chapters in Part II, "CLX." However, this chapter covers allocating and deallocating memory for objects. You will find this material in this chapter's coverage of pointers. In particular, you should see the sections "Working with Pointers to Objects" and "Pointers, Constructors, and Destructors." Included are a brief overview of the Pascal constructor and destructor syntax and a few words on writing virtual methods.
- **Interfaces**—Interfaces enable you to define the structure for a class without creating an implementation for it. I will cover this important topic in Part II.

# Integer Types and Floating-Point Types



I assume that readers of this book can quickly come to terms with basic Pascal types. I will say a few words on the basics of the subject and then go on to cover some issues that might trip up experienced programmers new to this language or experienced programmers who need a refresher course on Pascal syntax. This approach will not be helpful for a newcomer to programming, but it should be enough information to get experts up to speed on Object Pascal in short order.

I'll start by talking briefly about integers and floating-point types. After getting that basic material out of the way, I'll discuss strings, pointers, and typecasting.

Here are two very basic definitions:

- Integers are whole numbers, such as –1, 0, 1, 2, and 5,000,000.
- Floating-point numbers are sometimes known as decimal numbers, such as 7.0, 3.2, –5.004, and 32,000.0000000034.

## Ordinal Types

A whole series of types in Pascal are based on whole numbers. These include the `Byte`, `Integer`, `Char`, and `Boolean` types. All of these types are ordinal types.

Understanding the definition of the ordinal types is helpful for programmers who want to set sail on the good ship Pascal. All but the first and last members of ordinal types have a predecessor and a successor. For instance, an ordinal number such as 1 is followed by 2 and preceded by 0. The same cannot be said of a floating-point type. What is the predecessor to

1.0002? Is it 1.0001? Maybe. But perhaps it is 1.00019—or maybe 1.000199. How about 1.000199999999? Ultimately, there is no clearly defined predecessor to a floating-point number, so it is not an ordinal value.

Whole numbers are ordinal numbers. Simple types such as `Char` and `Boolean` are also ordinal numbers. For instance, the letter B is succeeded by the letter C and preceded by the letter A. It makes sense to talk about the successor and predecessor of a `Char`. The same is true of `Boolean` values. `False`, which is equivalent to 0, is succeeded by `True`, which is usually equivalent to 1. `False` is the predecessor of `True`, and `True` is the successor to `False`.

**NOTE**

When I use the word *integer* in a generic sense, I am talking about the numeric ordinal types such as `Byte`, `LongInt`, `Integer`, or `Cardinal`. When I talk specifically about `Integers`, with a capital I, then I mean the Pascal type named `Integer`. C and Java programmers make a similar distinction between the floating-point types and the type named `float`.

Two integer types exist in Pascal: generic and fundamental. The generic types, called `Integer` and `Cardinal`, will transform themselves to fit the compiler you are currently using. If you use `Integers` on a 16-bit platform, they will be 16 bits in size. Use them on a 32-bit platform with a 32-bit compiler, and they will be 32 bits in size. Use them on a 64-bit platform with a 64-bit compiler, and they will be 64 bits in size.

`Integers` are always signed values, which means that they use 1 bit to signal whether they are positive or negative and then use the remaining bits to record a number.

**NOTE**

Some readers might not be clear on the difference between signed and unsigned types. Consider the `Byte` and `ShortInt` types, both of which contain 8 bits. A `Byte` is unsigned, and a `ShortInt` is signed. The largest unsigned 8-bit number is 255, while its smallest value is 0. The largest signed 8-bit number is 127, while its smallest value is –128. The issue is simply that you can have 256 possible numbers that can be held in 8 bits. These 256 numbers can range from either –128 to 127, or from 0 to 255. The difference between signed and unsigned types is whether one of the bits is used to designate the plus and minus sign. Unsigned numbers have no plus and minus sign and, therefore, are always positive. Signed numbers range over both positive and negative values.

**NOTE**

At this time, there is only a 32-bit compiler in Kylix. On the Windows platform, at the time of this writing, there is a 16-bit and a 32-bit Delphi compiler.

In contrast to generic types, fundamental types are always a set size, regardless of what platform you use. For instance, a `Byte` is an 8-bit, unsigned number, regardless of the platform. The same rule applies to `LongInts`, which are always 32-bit signed numbers, regardless of the platform.

The generic types are shown in Table 3.1, and the fundamental types appear in Table 3.2.

**TABLE 3.1** The Generic Types Are Ordinal Values with a Range That Changes Depending on the Number of Bits in the Native Type Word for Your Platform

| *Type* | *Range* | *Format* |
|---|---|---|
| `Integer` | –2147483648 to 2147483647 | Signed 32-bit |
| `Cardinal` | 0 to 4294967295 | Unsigned 32-bit |

**TABLE 3.2** The Fundamental Type Stays the Same, Regardless of Platform

| *Type* | *Range* | *Format* |
|---|---|---|
| `ShortInt` | –128 to 127 | Signed 8-bit |
| `SmallInt` | –32768 to 32767 | Signed 16-bit |
| `LongInt` | –2147483648 to 2147483647 | Signed 32-bit |
| `Int64` | $-2^{63}$ to $2^{63-1}$ | Signed 64-bit |
| `Byte` | 0 to 255 | Unsigned 8-bit |
| `Word` | 0 to 65535 | Unsigned 16-bit |
| `LongWord` | 0 to 4294967295 | Unsigned 32-bit |

Most knowledgeable programmers try to use the generic types whenever possible. They help you port your code to new platforms, and they make your code backward compatible with old platforms. Furthermore, the generic `Integer` type should always be the fastest numeric type on any platform because it fits exactly the size of a word on that particular processor. Thus, the `Integer` type will usually be the best choice for producing fast code, even though it is larger than the `ShortInt or SmallInt` types. On 32-bit platforms, `LongInts` and `Integers` are

equally fast, but when we move to 64-bit computers, LongInts will no longer be the native type, while Integers will continue in the anointed position. In short, Integers will have 64 bits on a 64-bit platform, 16 bits on a 16-bit platform, and 32 bits on a 32-bit platform.

If you have code that assumes that a particular variable has a certain number of bits in it or will always contain only a certain range of numbers, you should use fundamental types to make sure that your code does not break if you move to another platform. Obviously, most of the code that you write will not be dependant on the number of bits in a variable, but if your code does depend on such a thing, choose your types carefully. For instance, if you are writing a routine that needs to handle numbers larger than 32,767, don't use Integers if you think that the code will ever be run on a 16-bit platform. If you do choose Integers, they will not be large enough to hold the values that you want to use. If you choose LongInts, the type will be large enough, even if ported to a 16-bit platform. (Of course, the odds that you will port your code back to a 16-bit platform are low.)

## Pascal Routines for Using Ordinal Numbers

The Integer types, by definition, are ordinal numbers. Ordinal numbers can be manipulated with the following routines: Ord, Pred, Succ, High, and Low. If applied to a Char, the Ord function returns its numeric value. For instance, the Ord of *A* is 65, and the Ord of a space is 32. In the following example, Num will be set to 66:

```
var
  MyChar: Char;
  Num: Integer;
begin
  MyChar := 'B';
  Num := Ord(MyChar);
end;
```

The Pred routine returns the predecessor of a number. For instance, the Pred of 1 is 0. The Succ of 1 is 2.

High and Low give you the highest and lowest numbers that you can use with a type. Examples of how to use High and Low are shown in the SimpleTypes program, found on your CD and in Listing 3.1.

**LISTING 3.1** The SimpleTypes Program

```
unit Main;

interface

uses
  SysUtils, Types, Classes,
```

**LISTING 3.1** Continued

```
  QGraphics, QControls, QForms,
  QDialogs, QStdCtrls, QExtCtrls;

type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    RadioGroup1: TRadioGroup;
    procedure RadioGroup1Click(Sender: TObject);
  private
    procedure DoLongInt;
    procedure DoInteger;
    procedure DoCardinal;
    procedure DoLongWord;
    procedure DoWord;
    procedure DoShortInt;
    procedure DoSmallInt;
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

type
  TMethodType = (mtInteger, mtCardinal, mtLongInt,
    mtLongWord, mtWord, mtShortInt, mtSmallInt);

procedure TForm1.DoInteger;
var
  Value: Integer;
begin
  ListBox1.Items.Add('Integer high value: ' + IntToStr(High(Value)));
  ListBox1.Items.Add('Integer low value: ' + IntToStr(Low(Value)));
  ListBox1.Items.Add('Size of Integer: ' + IntToStr(SizeOf(Value)) + ' bytes or
➥' + IntToStr(8 * SizeOf(Value)) + ' bits.');
end;

procedure TForm1.DoLongInt;
var
```

**LISTING 3.1** Continued

```
  Value: LongInt;
begin
  ListBox1.Items.Add('LongInt high value: ' + IntToStr(High(Value)));
  ListBox1.Items.Add('LongInt low value: ' + IntToStr(Low(Value)));
  ListBox1.Items.Add('Size of LongInt: ' + IntToStr(SizeOf(Value)) + ' bytes or
➥' + IntToStr(8 * SizeOf(Value)) + ' bits.');
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  case TMethodType(RadioGroup1.ItemIndex) of
    mtInteger: DoInteger;
    mtCardinal: DoCardinal;
    mtLongInt: DoLongInt;
    mtLongWord: DoLongWord;
    mtWord: DoWord;
    mtShortInt: DoShortInt;
    mtSmallInt: DoSmallInt;
  end;
end;

procedure TForm1.DoCardinal;
var
  Value: Cardinal;
begin
  ListBox1.Items.Add('Cardinal high value: ' + IntToStr(High(Value)));
  ListBox1.Items.Add('Cardinal low value: ' + IntToStr(Low(Value)));
  ListBox1.Items.Add('Size of Cardinal: ' + IntToStr(SizeOf(Value)) + ' bytes
➥or ' + IntToStr(8 * SizeOf(Value)) + ' bits.');
end;

procedure TForm1.DoLongWord;
var
  Value: LongWord;
begin
  ListBox1.Items.Add('LongWord high value: ' + IntToStr(High(Value)));
  ListBox1.Items.Add('LongWord low value: ' + IntToStr(Low(Value)));
  ListBox1.Items.Add('Size of LongWord: ' + IntToStr(SizeOf(Value)) + ' bytes
➥or ' + IntToStr(8 * SizeOf(Value)) + ' bits.');
end;

procedure TForm1.DoWord;
var
  Value: Word;
```

**LISTING 3.1** Continued

```
begin
  ListBox1.Items.Add('Word high value: ' + IntToStr(High(Value)));
  ListBox1.Items.Add('Word low value: ' + IntToStr(Low(Value)));
  ListBox1.Items.Add('Size of Word: ' + IntToStr(SizeOf(Value)) + ' bytes or '
➥+ IntToStr(8 * SizeOf(Value)) + ' bits.');
end;

procedure TForm1.DoShortInt;
var
  Value: ShortInt;
begin
  ListBox1.Items.Add('ShortInt high value: ' + IntToStr(High(Value)));
  ListBox1.Items.Add('ShortInt low value: ' + IntToStr(Low(Value)));
  ListBox1.Items.Add('Size of ShortInt: ' + IntToStr(SizeOf(Value)) + ' bytes
➥or ' + IntToStr(8 * SizeOf(Value)) + ' bits.');
end;

procedure TForm1.DoSmallInt;
var
  Value: SmallInt;
begin
  ListBox1.Items.Add('SmallInt high value: ' + IntToStr(High(Value)));
  ListBox1.Items.Add('SmallInt low value: ' + IntToStr(Low(Value)));
  ListBox1.Items.Add('Size of SmallInt: ' + IntToStr(SizeOf(Value)) + ' bytes
➥or ' + IntToStr(8 * SizeOf(Value)) + ' bits.');
end;

end.
```



This program rather laboriously calls `High` and `Low` for all the integer types. Notice that it also uses the `SizeOf` function, which returns the size in bytes of any variable or type. The point of this program is to show you that you can discover this information at runtime.

You can learn even more about a type using Run Time Type Information (RTTI). An introduction to RTTI appears in the upcoming section "Floating-Point Types."

## Enumerated Types

All major languages have enumerated types. This is an ordinal type. In fact, enumerated types are really nothing more than a few numbers starting from 0 and rarely ranging much higher than 10. The interesting thing about these numbers is that you can give them names.

Consider this example:

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TComputerLanguage = (clC, clCpp, clJava, clPascal, clVB);
var
  ComputerLanguage: TComputerLanguage;
begin
  ComputerLanguage := clPascal
end;
```

Here the values `clC`, `clCpp`, `clJava`, `clPascal`, and `clVB` are just fancy ways to write 0, 1, 2, 3, and 4. In short, enumerated types are just a way to associate numbers with names. However, if you want to associate numbers with names, you are always in danger of forgetting which number belongs to which name. For instance, you might want to reference Java and accidentally write 3, when what you meant to write was 2. To avoid confusion, the enumerated type enables you to associate a name with a number. Furthermore, you can enforce that relationship through Pascal's strong type checking. For instance, you can't assign even a valid identifier named `clTunaFish` to the variable `ComputerLanguage` unless it is part of the `TComputerLanguage` type.

**NOTE**

The letters `cl` prefacing each name are a Pascal convention. The convention says that you put the letters of the name of the type before the name. So, *C*omputer *L*anguage becomes `cl`.

You can use the `Ord` routine to convert an enumerated value to a number:

```
i := Ord(clPascal);
```

In this case, `i` is set equal to 3. In fact, you can go from the number to the name, but that is a complex operation involving routines found in the `TypInfo` unit. The `TypInfo` unit will be discussed in Chapter 4, and in the next section "Floating-Point Types."

Here is an example by Bob Swart that shows a simple way to write out the name of a type:

```
program BobEnum;
{$APPTYPE CONSOLE}
type
  TEnum = (zero, one, two, three);
var
  E: TEnum;
```

```
begin
  E := TEnum(2); // E := two;
  if E = two then WriteLn('Two!');
  ReadLn
end.
```

## Floating-Point Types

Pascal has lots of floating point-types for helping you work with decimal numbers, or fractions of whole numbers. Table 3.3 lists the fundamental types you can choose from.

**TABLE 3.3** The Fundamental Floating-Point Types—the Generic Type, Known as a `Real`, Is Currently Equivalent to a `Double`

| *Type* | *Range* |
|---|---|
| `Real48` | $2.9\times10^{-39}$ to $1.7\times10^{38}$ 11–12 6 |
| `Single` | $1.5\times10^{-45}$ to $3.4\times10^{38}$ 7–8 4 |
| `Double` | $5.0\times10^{-324}$ to $1.7\times10^{308}$ 15–16 8 |
| `Extended` | $3.6\times10^{-4951}$ to $1.1\times10^{4932}$ 19–20 10 |
| `Comp` | $-2^{63+1}$ to 2^63 –1 19–20 8 |
| `Currency` | –922337203685477.5808 to 922337203685477.5807 19–20 8 |

The most commonly used type is the `Double`. However, there is a generic floating point type known as a `Real`. It is currently the same as a `Double`, much as an `Integer` is currently the same as a `LongInt`. If you declare your floating-point values to be `Real`s, they can be automatically converted to any new optimal floating-point type that might come along.

The `Comp` type is the floating-point type that is used the least frequently. In fact, it isn't really meant to represent floating-point numbers. In the bad old days of 16-bit computing, this used to be the best way to work with very large whole numbers. Now it has no function other than to support old code. If you need to work with really large `Integer` types, then you should now use the `Int64` type.

**NOTE**

Back in the aforementioned bad old days, Pascal used to have a set of routines for working with a type of unique floating-point type known as a `Real`. (This is not the same thing as the current `Real` type, but it's a strange 48-bit beast that was custom-

made by optimization-obsessed Borland programmers. Back in this time, the `Real` type and its associated code were considered to be very fast. However, those days are now little more than a memory. Modern operating systems and modern processors now have built-in routines that are superior to the once–ground-breaking code that supported the 48-bit `Real` type.

The current `Real` is the same as a `Double`. However, a type known as a `Real48` is compatible with the old `Real` type used long ago, when Windows was a failed project that provided fodder for jokes and everyone believed that Apple might end up ruling the computer desktop. If you are an old Pascal programmer who has some code dependant on the implementation of the old Pascal `Real` type, then use `Real48`.

Remember that `Real` types, which are synonymous with the `Double` type, are now back in fashion. I'm having trouble adopting to this new state of affairs because I'm used to thinking of `Real`s as being out-of-date. So, you will find a lot of `Double`s in my code, but I am trying to make the move to using `Real`s instead. I never use `Real48`s because I have no code dependant on them.

## The `TBcd` and Floating-Point Accuracy

We now broach the treacherous topic of floating-point accuracy. This is a sea in which no ship is safe, and only caution can protect us from the reefs.

All experienced programmers know that floating-point types such as `Double`s and `Single`s lose precision nearly every time they are part of a calculation. This loss of precision is usually not a problem in a standard math or graphics program, but it can be a serious issue in financial calculations. As a result, you should consider using the `Currency` type to avoid rounding errors. However, this is not a perfect solution.

The best way to avoid problems with rounding errors is to use the `TBcd` type, found in the `FMTBcd` unit. *BCD* stands for "binary coded decimal," and it is a widely used technology to avoid rounding errors when working with floating-point numbers. Borland did not invent the BCD technology any more than it invented the idea of the floating-point type. It's just a technology that it employs in this product.

Here is what the `TBcd` type looks like:

```
  PBcd = ^TBcd;
  TBcd  = packed record
    Precision: Byte;                        { 1..64 }
    SignSpecialPlaces: Byte;                { Sign:1, Special:1, Places:6 }
    Fraction: packed array [0..31] of Byte; { BCD Nibbles, 00..99 per Byte,
high Nibble 1st }
  end;
```

Each of the numbers in your floating-point type is stored in a *nibble*, which is 4 bits in size. You can specify the number of digits in your number in the `Precision` field, and you can specify the number of places after the decimal in the `SignSpecialPlaces` field. In practice, you rarely end up working so directly with this type. Instead, you can use a series of routines to make working with the `TBcd` type a more palatable exercise.

Kylix provides a large number of routines in the `FMTBcd` unit for manipulating BCD values. A large sampling of these routines is found in Listing 3.2. You should find the time to open the unit itself and examine it as well.

**NOTE**

In the Kylix editor, if you put your cursor over any unit in your `uses` clause and then press Ctrl+Enter, the source for that unit should open in your editor.

**LISTING 3.2** Routines in `FMTBcd` That You Can Use to Help You Work with the BCD Type

```
procedure VarFMTBcdCreate(var ADest: Variant; const ABcd: TBcd); overload;
function VarFMTBcdCreate: Variant; overload;
function VarFMTBcdCreate(const AValue: string;
  Precision, Scale: Word): Variant; overload;
function VarFMTBcdCreate(const AValue: Double;
  Precision: Word = 18; Scale: Word = 4): Variant; overload;
function VarFMTBcdCreate(const ABcd: TBcd): Variant; overload;
function VarIsFMTBcd(const AValue: Variant): Boolean; overload;
function VarFMTBcd: TVarType;

// convert String/Double/Integer to BCD struct
function StrToBcd(const AValue: string): TBcd;
function TryStrToBcd(const AValue: string; var Bcd: TBcd): Boolean;
function DoubleToBcd(const AValue: Double): TBcd; overload;
procedure DoubleToBcd(const AValue: Double; var bcd: TBcd); overload;
function IntegerToBcd(const AValue: Integer): TBcd;
function VarToBcd(const AValue: Variant): TBcd;

function CurrToBCD(const Curr: Currency; var BCD: TBcd; Precision: Integer =
➥32;
  Decimals: Integer = 4): Boolean;

// Convert Bcd struct to string/Double/Integer
function BcdToStr(const Bcd: TBcd): string; overload;
function BcdToDouble(const Bcd: TBcd): Double;
```

**LISTING 3.2** Continued

```
  function BcdToInteger(const Bcd: TBcd; Truncate: Boolean = False): Integer;
  function BCDToCurr(const BCD: TBcd; var Curr: Currency): Boolean;

  // Formatting Bcd as string
  function BcdToStrF(const Bcd: TBcd; Format: TFloatFormat;
    const Precision, Digits: Integer): string;
  function FormatBcd(const Format: string; Bcd: TBcd): string;
  function BcdCompare(const bcd1, bcd2: TBcd): Integer;
```

Most of these routines are encapsulations of technologies for converting back and forth from `TBcd` values to most major types. For instance, `BcdToStr` converts a `TBcd` value to a string, and `StrToBcd` performs the opposite task. However, there are more routines than the ones I show you here. Perhaps the best way to get up to speed is to simply look at the BCDVariant program, found in Listing 3.3.

**LISTING 3.3** The BCDVariant Program Gives You a Number of Examples on How to Use the `BCD` Type

```
unit Main;

interface

uses
  SysUtils, Types, Classes,
  QGraphics, QControls, QForms,
  QDialogs, QStdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure SimpleMathButtonClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

**LISTING 3.3** Continued

```
var
  Form1: TForm1;

implementation

uses
  FMTBcd;

{$R *.xfm}

procedure TForm1.Button1Click(Sender: TObject);
var
  B: TBcd;
  V: Variant;
begin
  V := VarFMTBcdCreate(36383.530534346, 32, 9);
  ListBox1.Items.Add(BCDToStr(VarToBCD(V)));
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  B: TBcd;
  D: Double;
  V: Variant;
  S: String;
begin
  D := 32.346;
  V := VarFMTBcdCreate(D, 18, 3);
  B := VarToBcd(V);
  S := BcdToStr(B);
  ListBox1.Items.Add(S);
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  C: Currency;
  D: Double;
  B: TBcd;
  V: Variant;
  S: String;
begin
  C := 33334.43;
  CurrToBCD(C, B, 32, 4);
  D := BcdToDouble(B);
```

**LISTING 3.3** Continued

```
  S := Format('%m', [D]);
  ListBox1.Items.Add(S);
end;

procedure TForm1.SimpleMathButtonClick(Sender: TObject);
var
  V1, V2, V3: Variant;
  B1, B2, B3: TBcd;
  S1, S2: String;
begin
  V1 := VarFMTBcdCreate(3.5011, 32, 12);
  V2 := VarFMTBcdCreate(3.5020, 32, 12);
  V3 := V1 + V2;
  ListBox1.Items.Add(BcdToStr(VarToBcd(V3)));
  V3 := V1 * V2;
  ListBox1.Items.Add(BcdToStr(VarToBcd(V3)));
  V3 := V1 / V2;
  ListBox1.Items.Add(BcdToStr(VarToBcd(V3)));
  V3 := V1 - V2;
  ListBox1.Items.Add(BcdToStr(VarToBcd(V3)));
  B1 := VarToBcd(V1);
  B2 := VarToBcd(V2);
  S1 := BcdToStr(B1);
  S2 := BcdToStr(B2);
  BcdAdd(B1, B2, B3);
  ListBox1.Items.Add(S1 + ' + ' + S2 + '=' + BcdToStr(B3));
  BcdMultiply(B1, B2, B3);
  ListBox1.Items.Add(S1 + ' * ' + S2 + '=' + BcdToStr(B3));
  BcdDivide(B1, B2, B3);
  ListBox1.Items.Add(S1 + ' / ' + S2 + '='  + BcdToStr(B3));
  BcdSubtract(B1, B2, B3);
  ListBox1.Items.Add(S1 + ' - ' + S2 + '='  + BcdToStr(B3));
end;

end.
```

The most important code found here is seen in the lines at the beginning of the `SimpleMathButtonClick` method:

```
  V1 := VarFMTBcdCreate(3.5011, 32, 12);
  V2 := VarFMTBcdCreate(3.5020, 32, 12);
  V3 := V1 + V2;
  ListBox1.Items.Add(BcdToStr(VarToBcd(V3)));
  V3 := V1 * V2;
  ListBox1.Items.Add(BcdToStr(VarToBcd(V3)));
```

This code uses `VarFMTBcdCreate` to create two BCD numbers as `Variants`. Pass the number that you want to encapsulate in a BCD type in the first parameter. In the second parameter, pass the number of digits used to capture your number. For instance, in the first case shown previously, I could safely pass `5` because there are only five digits in 3.5011. Passing `32` is probably overkill. The last parameter is the number of those digits that appear after the decimal point. Again, `12` is more than ample for the job because there are only four numbers after the decimal point. (Obviously, these later two values are simply being used to fill in the `Precision` and `SignSpecialPlaces` fields of the `TBcd` type.)

The big question here is not how to call `VarFMTBcdCreate`, but why I am calling it. After all, this function returns not a `TBcd` value, but a `Variant`. I create `Variants` because you can directly add, multiply, divide, and subtract `TBcd` values when they are inside `Variants`:

```
V3 := V1 + V2;
```

Go back again to the declaration for `TBcd`. Clearly, there is no simple way to add or multiply values of type `TBcd`. However, if we convert them to `Variants`, the chore of performing basic math with `TBcd` values is marvelously simplified!



**NOTE**

For now, you need know little more than that `Variants` are like variables declared in a BASIC program, or like variables in Perl or Python. They are very loosely typed—or, at least, *appear* to be loosely typed. You can assign an `Integer`, `Real`, `Byte`, `String` or `Object`, to a `Variant`. In fact, you can assign almost anything to a `Variant`.

Kylix will allow you to implement particular kinds of `Variants`, and then give them interesting characteristics. For instance, you can create a kind of `Variant` that handles BCD values, and then you can teach this kind of `Variant` to do all sorts of interesting things In particular, you can teach it to handle addition, multiplication, and division. Clearly, this is the closest thing that Pascal has to the wonders of C++ operator overloading. If you are a Delphi programmer, you might find my definition of `Variants` at odds with what you learned about `Variants` in the Windows world. That is because `Variants` in Linux (and also in Delphi 6) do wondrous things that they did not do in the old Delphi 5 days.

Later in the `MathButtonClick` method, you see that there is a way to add, subtract, multiply, and divide `TBcd` variables without converting them to `Variants`:

```
BcdAdd(B1, B2, B3);
ListBox1.Items.Add(S1 + ' + ' + S2 + '=' + BcdToStr(B3));
BcdMultiply(B1, B2, B3);
```

```
ListBox1.Items.Add(S1 + ' * ' + S2 + '=' + BcdToStr(B3));
BcdDivide(B1, B2, B3);
ListBox1.Items.Add(S1 + ' / ' + S2 + '='  + BcdToStr(B3));
BcdSubtract(B1, B2, B3);
ListBox1.Items.Add(S1 + ' - ' + S2 + '='  + BcdToStr(B3));
```

This code shows you the `BcdAdd`, `BcdMultiply`, `BcdDivide`, and `BcdSubtract` routines, all of which do precisely what their names imply. However, most programmers would probably prefer the `Variant` code shown earlier because it is provides a more intuitive syntax.

Again, the point of the `TBcd` type is to ensure that you have no loss of precision when working with floating-point numbers. It goes without saying that there is overhead associated with the `TBcd` type and that, if possible, you should stick with `Doubles` or `Reals` if speed is an issue for you.

I should perhaps make clear that floating-point types are not inordinately inaccurate. The problems that you encounter with them occur when you do the kind of rounding necessary when working with money. If you don't have to round the values that you are working with to two decimal places, the standard floating-point types will probably meet your needs in all but the most rigorous of circumstances. More specifically, `Doubles` will generally be accurate to at least seven or eight decimal places, which in most cases is all the accuracy you will need. But if you keep rounding those values back to two decimal places, as you do when working with money, then the process of rounding the numbers will lead to errors of at least one penny.

## Pascal Strings

I've included this section on strings because this feature of the language has a number of very confusing aspects. Under normal circumstances, Pascal strings are very easy to use. However, there happen to be a number of different kinds of Pascal strings, and that proliferation of types really cries out for a clear explanation.

Object Pascal has four different kinds of strings: `ShortStrings`, `AnsiStrings`, `PChars`, and `WideStrings`. All Object Pascal strings except `WideStrings` are, at heart, little more than an array of `Char`. A `WideString` is an array of `WideChars`. A `Char` is 8 bits in size, while a `WideChar` is 16 bits—going on 32 bits—in size. I will explain more about `WideStrings` and `WideChars` at the end of this section on strings.

The following code fragment gives you examples of the types of things you can do with a `Char` or a `String`. The code explicitly uses `AnsiStrings`, but most of it would work the same regardless of whether the variables `S` and `S2` were declared as `ShortStrings`, `PChars`, or `AnsiStrings`. Of course, I will explain the differences among these three types later in this section. Here is the example:

```
var
  a, b: Char;
  S, S2: String;
begin
  S := 'Sam';     // Valid: Set a string equal to a string literal
  S := '1';       // Valid: Set a string equal to character
  S := '';        // Valid: Set a string equal to an empty string literal
  a := '1';       // Valid: Set a Char equal to a character literal
  b := a;          // Valid: Set a Char equal to Char
  a := 'Sam';     // Invalid: You can't set a Char equal to a string
  a := #65;       // Valid: Set a Char equal to a character literal
  a := Char(10);  // Valid: Set a Char equal to an integer converted to a char
  a := S[1];      // Valid: Set a Char equal to the first Char in a string
  S2 := 'Sam'#10; // Valid: Set a string equal to a string with Char appended
  S := S + S2;    // Valid: Concatenate two strings
  if (S = S2) then
    ShowMessage('S and S2 contain equivalent strings');
  if (S > S2) then
    ShowMessage('S would appear in a dictionary after S2');
end;
```

The Pascal language originated in Europe, so strings follow the traditional European syntax and are set off with single rather than double quotes. The code shown here declares two `Char`s and two `String`s. The first statement after the `begin` correctly sets the `String` equal to a string literal that contains three letters. You can also set a `String` equal to a string literal that contains a single character or no characters. You can set a `Char` equal to a single character such as a, b, A, or B. You cannot set a `Char` equal to a string such as Sam. You can, however, set a `Char` equal to the first character in a `String`, as in `a := S[1]`. You can also set a `String` equal to the 65th character in a character set by writing this syntax: `a := #65`. In the standard ANSI character set, the 65th character is a capital A, so this is equivalent to setting a `Char` equal to the letter A: `a := 'A';`. The expression `Char(10)` is equivalent to the expression `#10`. Both expressions reference the 10th ANSI character, which is usually the linefeed character. It is also legal to append or insert characters into a string using the following syntax: `S := 'Sam'#10;`. This adds a linefeed to the end of the string. Notice that the character is appended outside the closing quote.



**C/C++, JAVA NOTE**

In Java or C++ you would write `"Sam\n"` rather than `'Sam'#10`. The two statements are equivalent.

Studying the examples in this section should give you some sense of how to use strings in your programs. Notice that in one of the examples, you can use the + operator to concatenate two strings. You can also use the < and > operators to test whether a `String` is larger than another `String`, and you can use the = operator to test whether two `Strings` point to identical sets of characters.

**JAVA NOTE**

The = operator in Pascal does the same thing as the `String::equals` method does in Java. You are not testing to see whether the strings point at the same memory; you are testing to see whether they point at strings that contain the same sets of characters.

## ShortStrings

The `ShortString` is the oldest kind of Pascal string, and it is rarely in use today. A `ShortString` is essentially a glorified array of `Char` with a maximum length of 256 characters. The first byte, the *length byte*, designates the length of the string. `ShortStrings` are not null-terminated; their length is determined only by the length byte. Remember that the length byte takes up 1 of the 256 bytes in the string, so the longest possible `ShortString` contains 255 characters. The limitation on the length of a `ShortString` exists because the first byte is 8 bits in size, and you can fit only 256 possible values in 8 bits.

**NOTE**

`ShortStrings` are used mostly for backward compatibility with old Pascal code. However, you might use a `ShortString` if you need to be sure that a block of memory has a prescribed size. For instance, you know that `ShortStrings` are usually 256 bytes long, so if you want to create an array of 4 `Strings` and you want to be sure that it occupies exactly 1,024 bytes of memory, regardless of the length of each string (and assuming that each string is 255 characters in length or less), you might decide to use `ShortStrings` rather than `AnsiStrings`. `ShortStrings` can also be useful in variant `records`, as described in the later section of this chapter titled "Variant `Records`."

Here is the syntax for using a `ShortString`:

```
var
  S: ShortString;
```

```
begin
  S := 'Hello';
end;
```

This string is represented in memory as such: [#5][H][e][l][l][o]. The first byte of the string, which the user never sees, represents the length of the string. The remaining bytes contain the string itself.

You can also declare a `ShortString` like this:

```
var
  S: String[10];
```

This string contains only 10 characters rather than 255. More commonly, you might declare a type of string that is a custom length and then reuse that type throughout your program:

```
type
  String5 = String[5];
  String15 = String[15]
var
  S5: String5;
  S15: String15;
```



The compiler appears not to object to you assigning strings larger than 5 or 16 characters to the types declared previously. However, the string that you create will display only the appropriate number of characters. The others will be ignored.

Again, I want to stress that `ShortStrings` are not in common use today. In Java parlance, one might even say that they are *deprecated*, although I doubt that they will ever cease to be a part of the language.

## AnsiStrings

`AnsiStrings` are also known as *long strings*. On 32-bit platforms, the maximum length for an `AnsiString` is 2GB. This type is the native Object Pascal string and the kind that you will use in most programs.

If you declare a variable as a `String`, it is assumed to be an `AnsiString`. In other words, if you do not specify that a string is an `AnsiString`, a `ShortString`, or a custom string such as `String[10]`, you can assume that it is an `AnsiString`. The one exception to this rule occurs if you explicitly turn off the `$H` directive, where `H` can be thought of as standing for "huge" strings. In such cases, all strings are assumed to be `ShortStrings` unless explicitly declared otherwise. If you place the `{$H-}` directive at the top of a module, that entire module will use `ShortStrings` by default. If you deselect Project, Options, Compiler, Huge Strings from the menu, your entire program will use `ShortStrings` by default.

**NOTE**

When using the default key mappings, you can press Ctrl+O+O (that's the letter O) to get a list of all the compiler directives for the current module.

When a CLX method needs to be passed a string, it almost always expects to be passed an `AnsiString`. The `AnsiString` is the native type expected by CLX controls. Despite the simplicity of this statement, there are some twists and turns to it. As a result, I will discuss this in more depth both in this section and in the section "`PChars`."

An `AnsiString` is a pointer type, although you should rarely, if ever, need to explicitly allocate memory for it. The compiler notes the times when you make an assignment to a string, and it calls routines at that time for allocating the memory for the string. (Many of these routines are in System.pas, and you can step right into them with the compiler on some versions of Kylix.)

**NOTE**

You will find that many of the routines in the `System` unit use Assembly language. In general, they follow one of two different formats:

```
procedure Foo;
asm
  mov eax, 1
end;

procedure FooBar;
var
  X: Integer;
begin
  X := 7;
  asm
    mov eax, X;
  end;
end;
```

Procedure `Foo` uses `asm` where a normal Pascal procedure would use `begin`. In this type of procedure, all the code is written in Assembler until the closing `end` statement. The second example embeds an `asm` statement in a `begin..end` block. Both syntaxes are valid. When using the debugger, after starting your program, choose View, Debug Windows, CPU to step through your code. I will talk more about debugging in Chapter 5. However, I am not going to say anything more about Assembler in this book. Use System.pas as a reference if you are interested in this technology.

The only time that you might need to allocate memory for an `AnsiString` is if you are going to pass it to a routine that does not know about `AnsiStrings`—that is, when you are passing it a routine written in some language other than Pascal or when you are passing it to some exceptionally peculiar Pascal routine. In such a case, you would normally want to pass a `PChar` rather than an `AnsiString`. But it is possible to pass an `AnsiString` to such a routine; you allocate memory for it first and then pass it. (Use the `SetLength` routine to allocate memory for an `AnsiString`, as described at the very end of this section.)

Routines that take `PChars` are generally routines that are written in some other language, such as C or C++. If you pass an `AnsiString` into such a routine and you expect it to pass the string back with a new value in it (passing by reference), you probably need to allocate memory for the string before passing it. If you are passing an `AnsiString` into an Object Pascal routine, you can assume that the compiler will know how to allocate memory for it. In your day-to-day practice as an Object Pascal programmer, you should never need to think about allocating memory for an `AnsiString`. The cases when you need to do it are very rare and are not the type that beginning or intermediate-level programmers are ever likely to encounter.

`AnsiStrings` are null-terminated. This means that the end of the string is marked with `#0`, the first character in the ANSI character set. This is the same way that you mark the end of a string in C/C++. `AnsiStrings` are different than C/C++ strings, however, because they are usually prefaced with two 32-bit characters; one character holds the length of the string, and the other holds the reference count for a string. The only time that an `AnsiString` is not prefaced by these values is when the string variable references a 0-length string. As a programmer, you will almost certainly never have an occasion to explicitly reference either of these values.



It is a simple matter to understand the 32-bit value that holds the length of the string. It is similar to the length byte in a `ShortString`, except that it is 32 bits in size rather than 8 bits, so it can reference a very large string. What is the point, though, of the 32-bit value used for reference counting?

Reference counting is a means of saving memory and decreasing the time necessary to make string assignments. If two strings contain the same values, it is thriftiest to have them both point at the same memory. If possible, Object Pascal will do this by default. (You can override this behavior, as explained later in this section in the note on the `UniqueString` procedure.) When reference counting, the compiler simply points a second string at the memory allocated for a first string and then ups the reference count of the strings. Consider the following code fragment:

```
var
  Sam: String;
  Fred: String;
begin
```

```
  Sam := 'Look at all beings with the eyes of compassion. -- Lotus Sutra';
  Fred := Sam; // Reference count incremented, no memory allocated for chars.
  Fred := 'Learn to ' + Fred; // Strings not equal, memory must be allocated.
end;
```

When you set `Sam` equal to the quote from the Lotus Sutra, the compiler allocates sufficient memory for the variable `Sam`. When you set `Fred` equal to `Sam`, no new memory for character values is allocated. Instead, the reference count for the string is incremented and `Fred` is pointed at the same string as `Sam`. This kind of assignment is very fast and also saves memory. In short, you avoid both the extra memory consumed by allocating memory for the characters in the string and also the extra time required to copy the memory from one location in memory to another.

So far, so good. But what happens if you change one of the values that either variable addresses? That is what happens in the third line of the code fragment. When you change the value of `Fred` in the last line of the method, new memory is allocated for `Fred` and the reference count for the string is decremented by 1. At this point, `Fred` and `Sam` point at two entirely separate strings.

**NOTE**

You can use the `UniqueString` procedure to force a string to have a reference count of 1, even if it would normally have a higher count.

I want to stress that all these complicated machinations mean that you normally don't have to think about string memory allocation at all. You can just use a string type in a manner similar to the way you would use an `Integer` type. The compiler handles the allocation, and you don't have to think about it. However, it helps to know the inner workings of the `AnsiString` type, both so that you know what happens in unusual cases and so that you can design your code to be as efficient as possible.

Strings are generally allocated for you automatically. However, you can use the `SetLength` procedure to set or reset the length of a string:

```
var
  S: string;
begin
  SetLength(S, 10);
  SetLength(S, 12);
end;
```

Many routines built into the Object Pascal language can help you work with strings. In particular, see the `FmtStr` and `Format` functions. You might also want to browse the entire `SysUtils` unit and become familiar with the many useful routines found there. Also see the `LCodeBox` unit that ships with this book.

## PChars

A `PChar` is a standard null-terminated string and is structurally exactly like a C string. In fact, this type was created primarily to provide compatibility with C class libraries. In particular, it was created for compatibility with the Windows API, which is written in C. It has proven to be a generally useful type, and it will come in handy when you are calling functions from the Linux C libraries such as `Libc`.

**NOTE**

To call most of the routines in the `Libc` library, just add `Libc` to your `uses` clause and go to work. This process is described in more depth in Chapter 6, "Understanding the Linux Environment."



The native Object Pascal string type is known as a `String`—or, more properly, as a long string or `AnsiString`. However, in most cases you are free to use either the native `String` type or the `PChar` type. Both types of strings are null-terminated. The difference between them is that a Pascal string has data placed in front of the `String` that determines the string's length and its reference count.

In most cases in a Kylix program, you should use the `AnsiString` type. A Kylix control such as a `TEdit` would never expect you to pass it a `PChar`. However, it is usually legal, but unorthodox, to pass it a `PChar`. This is confusing enough that an example might be helpful. Consider the following block of code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Sam: PChar;
begin
  Sam := 'Fred';
  Edit1.Text := Sam;
end;
```

This code will compile and run without error. In short, it is legal to assign a `PChar` to a property that is declared to be of type `AnsiString`. (Actually the `Text` property is declared to be of type `TCaption`, but `TCaption` is declared to be of type `String`.)

**NOTE**

CLX is built on top of the C++ library called Qt. As a result, many of the controls in CLX ultimately end up working with native C strings, or a C `String` object. However, none of that is any concern to us as Pascal programmers. CLX is expecting `AnsiStrings` and, when you work with CLX controls, you should use the native `String` type.

You can assign a `PChar` to a string directly. However, if you assign a `String` to a `PChar`, you need to typecast it:

```
var
  S: string;
  P: PChar;
begin
  P := PChar(S);
```

As you recall, an `AnsiString` is simply a `PChar` with some data in front of it. This data appears at a negative offset from the pointer to the `AnsiString`. As a result, typecasting the `AnsiString` as a `PChar` is really just a confirmation that *from the pointer to the `AnsiString` and onward*, an `AnsiString` is nothing more than a `PChar`. You will use this typecasting technique quite often if you need to pass `AnsiStrings` to routines written in C that are expecting a regular C string rather than an `AnsiString`.

Once the decision was made to make PChars part of Object Pascal there needed to be a set of routines to help you work with such strings. These routines are based closely on the functions you would use for manipulating strings in a C/C++ program. For instance, these routines have names such as `StrLen`, `StrCat`, `StrPos`, and `StrScan`. Again, you should look in the `SysUtils` unit for more information on these routines. You will find that there are dozens of such routines and that they are quite flexible and powerful.

## WideStrings

`WideStrings` are very much like `AnsiStrings`, except that they point at wide characters of 16 bits rather than normal `Chars` of 8 bits. These large characters, known as `WideChars`, are a means of manipulating Unicode characters. Unicode in particular, and `WideChars`, in general, provide a means for working with large character sets that will not fit in the 256 bits of a `Char`. For instance, the kanji character sets from Asia have thousands of characters in them. You can't capture them using standard `AnsiStrings`; instead, you must use `WideStrings`.

**NOTE**

In Windows, the native wide character type (`WCHAR`) is 16 bits in size. In Linux, wide characters are 32 bits in size. The Kylix team decided to reuse the 16 bit `WideChar` in place for Windows rather than to rewrite the routines explicitly for the 32-bit Linux `WideChar`. As a result, your programs work with 16-bit `WideChars`, even though Linux defaults to 32-bit `WideChars`. Unless we are invaded from Alpha Centauri, where very large character sets are in common use, you should find that 16-bit `WideChars` are large enough for all practical purposes.

Starting with Kylix and Delphi 6, `WideStrings` are reference counted just as `AnsiStrings` are reference counted. In fact, you use a `WideString` exactly as you would use an `AnsiString`:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S: WideString;
begin
  S := 'Sam';
  Edit1.Text := S;
end;
```

This example shows that you can convert an `AnsiString` to a `WideString` and also convert a `WideString` to an `AnsiString` through the simple use of the assignment operator. In Kylix and Delphi 6, code based on `WideStrings` is actually quite efficient. If you have good reason to use `WideStrings`, go ahead and use them. The compiler handles them quite easily.

This is the end of the section on `Strings`. Next up are typecasts, a technology used very widely in Kylix programs. After that, we will look at the array and record types, and then we'll take a quick tour of Object Pascal pointers.

# Typecasts

Typecasts can be used to coerce a type declared one way to be treated as some other type that descends from it. For instance, in the following code a typecast is used to enable an object declared as type `TPersistent` to be treated as a `TStringList`:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TPersistent;
  L: TStringList;
begin
  L := TStringList(F);
end;
```

You perform a typecast by stating the name of the type that you want to declare and then placing the object that you want to typecast in parenthesis. The compiler is smart enough to know that this is a typecast, not a function call. The user can often make the same deduction simply by seeing that `TStringList` begins with the telltale `T`, which stands for "type." That particular initial consonant gives away the fact that `TStringList` is a type, not a function.

## The `as` and `is` Operators and the `Sender` Parameter

Typecasts are an essential part of working with a language that supports polymorphism. In many Kylix programs, the IDE generates event handlers that are passed objects declared to be one type but that are actually descendants of the type passed. The classic example of this is the `Sender` parameter, seen in many event handlers:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

The `Button1Click` method in a typical Delphi application is called when the user clicks on Button1. In that case, the `Sender` parameter really contains a `TButton` object. Flexibility is gained, however, if you declare the parameter as a `TObject`. The declaration enables you to pass any Pascal object to this method, not just a `TButton`.

**NOTE**

All Object Pascal classes must descend from `TObject` or one of its descendants. Even if you declare an object to have no ancestor, the Pascal compiler automatically descends the class from `TObject`. You simply can't declare an Object Pascal class that does not descend from `TObject` or one of its descendants. As a result of this rule, it is legal to pass any object in the `Sender` parameter of the `Button1Click` method. This extraordinary flexibility is possible because all these objects, by definition, descend from `TObject`.

Let's look at a practical example of how declaring the `Button1Click` method to take a `TObject` might be useful in a real program. Imagine that a message handler that you want to respond to clicks on not only a `TButton`, but also a `TBitButton` and `TSpeedButton`. The previous declaration could work with any of those types because they are all descendants of `TObject`. More particularly, the magic of polymorphism grants `TButton`, `TBitButton`, or `TSpeedButton` the capability to masquerade as a lowly `TObject`.

**NOTE**

Never forget the hierarchical nature of polymorphism: The king can masquerade as a pawn, but a pawn cannot pretend to be a king. A `TButton` can go disguised as a `TObject`, but a `TObject` cannot put on airs and pretend to be a `TButton`.

Here is how you can get at the object passed in the `Sender` parameter:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Button: TButton;
begin
  Button := TButton(Sender);
end;
```

As you can see, you need do nothing more than perform a simple typecast, and the `Sender` object is uncloaked and its true nature as an instance of `TButton` is revealed. In this case, the typecast again plays the role of the truth-teller, stripping off the object's disguise.

But doesn't trouble lurk amid the players of this game of cat and mouse? What would happen if the truth-teller were wrong? If you typecast a `Sender` object as a `TButton`, but it was really a `TBitButton`, `TSpeedButton`, `TPanel`, or something else passed unexpectedly, your typecast would be allowed at compile time, but an exception would be raised at runtime. Here is what you can do in such cases:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Button: TButton;
begin
  if Sender is TButton then
    Button := TButton(Sender);
end;
```

This code uses the `is` operator to test whether `Sender` is of type `TButton`. If it is, the `if` statement evaluates to `true` and the call succeeds. If it is not, the second line attempting the typecast is never called.

There is another way to perform a typecast that is perhaps a bit more modern than the way I have shown you so far:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Button: TButton;
begin
  Button := Sender as TButton;
end;
```

This code yields that same result as writing `Button := TButton(Sender)`. The difference is that the code will raise an exception if `Sender` is not of type `TButton`. The `as` operator is a prophylactic ensuring that the typecast will not be made if it is not valid. I also find the code easy to read because it cannot be mistaken as a method call, per the discussion earlier in this section. On the other hand, the code might take longer to execute than the other kind of typecast because it is adding the overhead of type checking. (An actual ruling on the performance of the two forms of typecasts will have to be left to someone else because I am not an expert in this kind of performance issue. But certainly the additional type checking must add at least some overhead to your program.)

The `as` operator is most typically used in statements that look like this:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TPersistent;
  L: TStringList;
begin
  (F as TStringList).Add('Sam');
end;
```

Here you can see the way parentheses have been added to give precedence to the `as` operator and ensure that the compiler can make proper sense of your statement.

Hopefully you now understand the basics of how typecasts work in an Object Pascal program. If you still feel that you could use more enlightenment on this issue, just be patient—the subject will come up numerous times throughout the course of this book.

# Arrays

Pascal is a very clean and easy-to-read language. If you can write code at all, you should be able to understand the basics of the Pascal syntax. The Pascal array syntax is no exception to this rule. Artfully constructed and robustly engineered, Pascal arrays are a powerful feature of the language.

Here is how to declare an array of `Integers` in Pascal:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyArray: array [0..10] of Integer;
  MyNum: Integer;
begin
  MyArray[0] := 1;
  MyNum := MyArray[0];
end;
```

In this code, you find an array of 11 `Integers`. The first member of the array is at offset `0`, and the last is at offset `10`. The syntax for capturing this construction is simplicity itself. It is hard to imagine how the concept of an array could be expressed more elegantly or more clearly.

Pascal arrays are very flexible. Note that you can declare not only the top of the range of values in an array, but also the bottom:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyArray: array [12..24] of Integer;
  MyNum: Integer;
begin
  MyArray[12] := 1;
  MyNum := MyArray[12];
end;
```

In this example, the first element of the array is at offset `12`, and the last is at `24`. You can discover at runtime the range of values in an array:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyArray: array [12..24] of Integer;
  HighValue, LowValue: Integer;
begin
  HighValue := High(MyArray);
  LowValue := Low(MyArray);
  Label1.Caption := 'High Value: ' + IntToStr(HighValue);
  Label2.Caption := 'Low Value: ' + IntToStr(LowValue);
end;
```

The `High` function returns the upper limit of an array, which, in this case, is `24`. `Low` returns the low range, which, in this case, is `12`.

## Array Constants

When I pick up a new language, I always seem to struggle to find an example of how to declare array constants. This next sample shows two array constants, one with `Integers` and the other with `Strings`:

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TMyStringArray = array[0..2] of String;
const
  MyIntArray: array[0..3] of Integer = (1, 2, 3, 4);
  MyStringArray: TMyStringArray = ('One', 'Two', 'Three');
```

```
begin
  ListBox1.Items.Add(IntToStr(MyIntArray[0])); // prints 1
  ListBox1.Items.Add(MyStringArray[0]); // prints 'one'
end;
```

In this example, I declare the type of the `Integer` array in the `const` statement and the type of the `String` array in a type section. I do this simply to show that, depending on your needs or preferences, you can use either syntax.

## Dynamic Arrays and Arrays of Objects

Unlike C/C++, Pascal has no trouble handling arrays of objects. The following syntax discovers all the buttons on a form, places them in an array, and gives you access to their properties and methods:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  ButtonArray: array of TButton;
  i, j, Num: Integer;
begin
  Num := 0;
  for i := 0 to ComponentCount - 1 do
    if Components[i] is TButton then
      Inc(Num);

  SetLength(ButtonArray, Num);

  j := 0;
  for i := 0 to ComponentCount - 1 do
    if Components[i] is TButton then begin
      ButtonArray[j] := TButton(Components[i]);
      Inc(j);
    end;

  for i := 0 to High(ButtonArray) do
    ListBox1.Items.Add(ButtonArray[i].Caption);
end;
```

The `Button3Click` method shows the Pascal language doing all sorts of glorious thing to delight and amuse us. The code discovers all the `TButton` objects on a form, declares an array just large enough to hold these buttons, and then places the buttons in the array. Finally, the captions of each button are placed in a list box, as shown in Figure 3.1.

`ButtonArray` is declared to be an array of `TButton`, with no declared bottom or top range. Instead, the range of the array will be discovered at runtime.

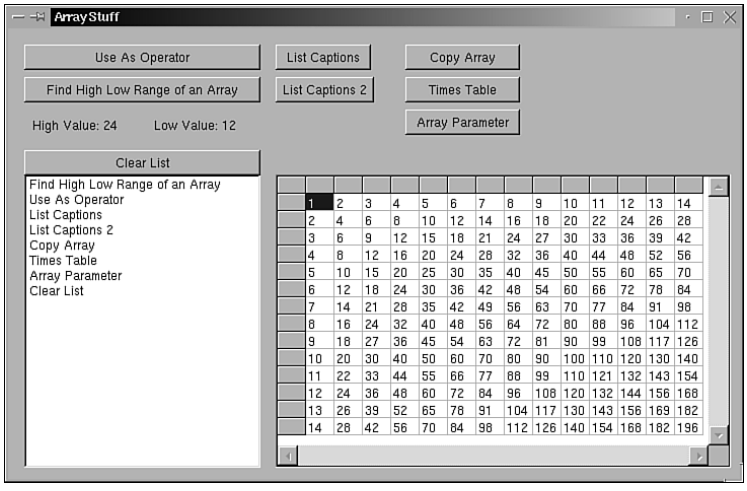


**FIGURE 3.1**

*A list box is put to use holding the captions of each button placed on this form.*

To find out how many elements we need in the array, the code makes use of information maintained by the main form of this application. In particular, all forms maintain a list of the components that have been dropped on them.

The program uses the `Form1.ComponentCount` property to determine how many components are on the form:

```
for i := 0 to ComponentCount - 1 do
```

Then each element of the array of components maintained by the main form is iterated over and is checked to see if it is a button:

```
if Components[i] is TButton then
```

Perhaps some clarity could be gained if these methods were written thus:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  ButtonArray: array of TButton;
  i, j, Num: Integer;
begin
  Num := 0;
  for i := 0 to Self.ComponentCount - 1 do
    if Self.Components[i] is TButton then
      Inc(Num);
```

In this rewrite of the first lines of the `Button3Click` method, I make explicit the fact that `ComponentCount` and the `Components` array both belong to `Form1`.

**C/C++, JAVA NOTE**

Object Pascal uses the term `Self` where Java or C++ would use `this`. Depending on your frame of mind, this personification of objects could be appealing or distracting. Despite the different flavors associated with the two terms, their meaning is ultimately identical.

When you know the number of buttons that were dropped on the form, you can set the size of the array of buttons:

```
SetLength(ButtonArray, Num);
```

The `SetLength` method does exactly what you would expect: It allocates memory in the array for the number of buttons that you have dropped on the form.

**NOTE**

You will perhaps recall that `SetLength` can also be used to set the length as an `AnsiString`. Because a string is really just an array of `Char`, this is not quite the coincidence that it might appear at first. Other functions that you can use on both types include `Copy` and `Length`.

The next chunk of code simply iterates over the components again, adding the buttons into the array as each is discovered:

```
for i := 0 to ComponentCount - 1 do
  if Components[i] is TButton then begin
    ButtonArray[j] := TButton(Components[i]);
    Inc(J);
  end;
```

Finally, the code displays the names of the buttons in a list box:

```
for i := 0 to High(ButtonArray) do
  ListBox1.Items.Add(ButtonArray[i].Caption);
```

Note that the syntax for accessing the members of the array of buttons is utterly clean and intuitive:

```
ButtonArray[i].Caption
```

It turns out that the `SetLength` procedure can be used not only to allocate memory for a `String`, but also to reallocate it. The following rewrite of the `Button3Click` method shows how to put this to use:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  ButtonArray: array of TButton;
  i, Num: Integer;
begin
  Num := 0;
  for i := 0 to Self.ComponentCount - 1 do
    if Self.Components[i] is TButton then begin
      Inc(Num);
      SetLength(ButtonArray, Num);
      ButtonArray[Num - 1] := TButton(Components[i]);
    end;

  for i := 0 to High(ButtonArray) do
    ListBox1.Items.Add(ButtonArray[i].Caption);
end;
```

Here the method is shortened and cleaned up a bit by continually reallocating the memory for the `ButtonArray` with the `SetLength` method.



**NOTE**

You might expect that repeatedly reallocating memory has some overhead associated with it, so it is possible that this second solution is not necessarily any faster than the first technique. People who are interested in such matters can explore the matter further on their own. Unless you are using these methods in a large, frequently called loop, don't worry about using one method or the other—they both execute in essentially zero time.

If you want to shorten the length of a dynamic array, use the `Copy` function:

```
procedure TForm1.Button5Click(Sender: TObject);
const
  BigLen = 24;
  SmallLen = 12;
```

```
var
  MyArray: array of Double;
  i: Integer;
begin
  SetLength(MyArray, BigLen);
  for i := 0 to High(MyArray) do
    MyArray[i] := Sqr(i);

  MyArray := Copy(MyArray, 0, SmallLen);

  for i := 0 to High(MyArray) do
    ListBox1.Items.Add(Format('Value of array = %f', [MyArray[i]]));
end;
```

This code first sets the length of `MyArray` to `24`. Next it assigns values to each member of the array. The array is then shortened to length `12`. Finally, the program displays the elements of the array to show that the shortening did not destroy the values that it held. Note that you can use `High`, `Low` and `Length` on dynamic arrays, just as you can on normal arrays and on strings.

Pascal arrays can contain multiple dimensions. Here is an example of how multi-dimensional arrays works:

```
procedure TForm1.Button6Click(Sender: TObject);
const
  XSize = 5;
  YSize = 5;
var
  MyTwoDim: array[0..XSize, 0..YSize] of Integer;
  i, j: Integer;
begin
  for i := 0 to YSize do
    for j := 0 to XSize do
      MyTwoDim[j, i] := j * i;

  for i := 0 to YSize do
    for j := 0 to XSize do
      StringGrid1.Cells[j, i] := IntToStr(MyTwoDim[j, i]);
end;
```

The code declares a two-dimensional array of `Integer`. Both dimensions contain five elements. The code then fills out the array with the first five elements of the multiplication table. A `TStringGrid` allows the program to display the content of the array, as shown in Figure 3.2. The key to the `TStringGrid` object is the `Cells` property, which represents the array of cells in the grid. The syntax for using the `Cells` property is the same as for using a two-dimensional array of `String`.

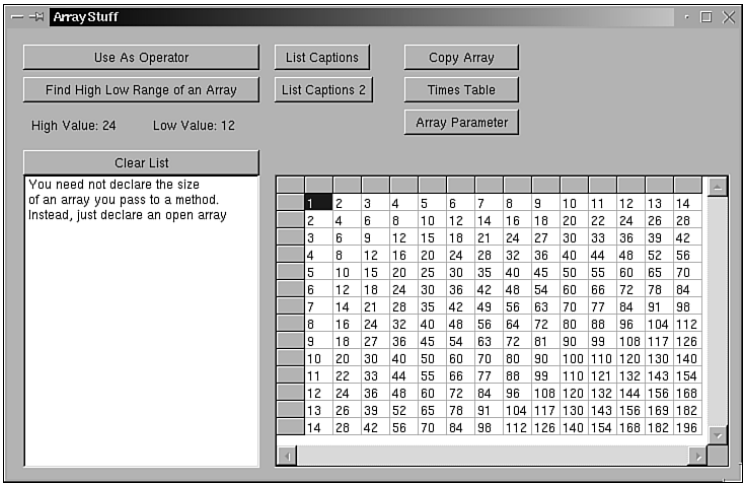


**FIGURE 3.2**

*The* `TStringGrid` *object from the Additional page in the Component Palette can help you display your data in a clean and logical fashion.*



You may declare an array in the normal fashion and then pass it to a method as a parameter. In such cases, you need not specify the size of the parameter in the function that you pass to it:

```
procedure TForm1.ShowArray(MyArray: array of String);
var
  i: Integer;
begin
  for i := 0 to High(MyArray) do
    ListBox1.Items.Add(MyArray[i]);
end;

procedure TForm1.Button7Click(Sender: TObject);
var
  MyArray: array [0..2] of String;
begin
  MyArray[0] := 'You need not declare the size';
  MyArray[1] := 'of an array you pass to a method.';
  MyArray[2] := 'Instead, just declare an open array';
  ShowArray(MyArray);
end;
```

Here you can see that I do not explicitly state the size of the array to be passed to the `ShowArray` method. Instead, I declare it as an open array of `String` and then use the `High` method to determine its actual dimensions.

## Debug Your Arrays: Turn on Range Checking

When developing applications that use arrays, it is best to select Project, Options, Compiler and turn on Range Checking. Range Checking is off, by default. This is a technique used by the compiler to check that you are not trying to write past the end of an array. For instance, if you allocate an array of 10 bytes in size and try to write to the nonexistent 11th byte of the array, the Range Checking mechanism would catch the error and raise an exception. Without range checking, sometimes writing past the end of an array will raise an exception; at other times, it will corrupt the memory of a program in some subtle and seemingly undetectable manner. This can lead to debugging sessions that last for hours or even days. If you turn on Range Checking, however, the error will appear immediately, and it will be clearly flagged. After debugging your code that uses arrays, you should turn off Range Checking because it slows down your code.

In this section on arrays, I discussed the basic facts about arrays in a few short paragraphs and then covered the more complex subject of dynamic arrays and open arrays. This look at arrays ended with a brief meditation on the importance of range checking.

# Records

*Records* are a technique for binding together multiple data types into a single structure. Examples of using records can be found in the RecordWorks program found in the Chap03 directory of your accompanying CD.

Consider the following method:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FirstName: String;
  LastName: String;
  Address: String;
  City: String;
  State: String;
  Zip: String;
begin
  FirstName := 'Blaise';
  LastName := 'Pascal';
  Address := '1623 Geometry Lane';
  City := 'Clemont';
  State := 'CA';
  Zip := '81662'
end;
```

The `String` declarations in the `var` section are all part of an address. Here is how you can bind these identifiers into one record:

```
procedure TForm1.Button2Click(Sender: TObject);
type
  TAddress = record
    FirstName: String;
    LastName: String;
    Address: String;
    City: String;
    State: String;
    Zip: String;
  end;
var
  Address: TAddress;
begin
  Address.FirstName := 'Blaise';
  Address.LastName := 'Pascal';
  Address.Address := '1623 Geometry Lane';
  Address.City := 'Clemont';
  Address.State := 'CA';
  Address.Zip := '81662'
end;
```

In this second example, a `record` declaration has been added to the `type` section of the `Button2Click` method. A variable of type `TAddress` has been declared in the `var` section. Dot notation is used in the body of the `Button2Click` method to enable you to reference each field of the record: `Address.FirstName`. People often will read out loud such a declaration as "Address dot FirstName," hence the term *dot notation*.



**C/C++ NOTE**

A record in Object Pascal is the same thing as a struct in C/C++. However, in C/C++, the line between a struct and an object is blurred, while in Object Pascal they are two distinctly different types. Pascal records do not support the concept of a method, although you can have a pointer to a method or function as a member of a record. However, that pointer references a routine outside the record itself. A method, on the other hand, is a member of an object.

## Records and `with` Statements

You can create a `with` statement to avoid the necessity of using dot notation to reference the fields of your record. Consider the following code fragment:

```
procedure TForm1.Button3Click(Sender: TObject);
type
```

```
  TPerson = Record
    Age: Integer;
    Name: String;
  end;
var
  Person: TPerson;
begin
  with Person do begin
    Age := 100046;
    Name := 'ET';
  end;
end;
```

The line `with Person do begin` is a `with` statement that tells the compiler that the identifiers `Age` and `Name` belong to the `TPerson` record. In effect, `with` statements offer a kind of shorthand to make it easier to use records or objects.

**NOTE**

`with` statements are a double-edged sword. If used properly, they can help you write cleaner code that's easier to understand. If used improperly, they can help you write code that not even you can understand. Frankly, I tend to avoid the syntax because it can be so confusing. In particular, the `with` syntax can break the connection between a record or object and the fields of that record or object.

For instance, you have to use a bit of reasoning to realize that `Age` belongs to the `TPerson` record and is not, for instance, simply a globally scoped variable of type `Integer`. In this case, the true state of affairs is clear enough, but in longer, more complex programs, it can be hard to see what has happened. A good rule of thumb might be to use `with` statements if they help you write clean code and then to avoid them if you are simply trying to save time by cutting down on the number of keystrokes you type.

## Variant `Records`

A variant `record` in Pascal enables you to assign different field types to the same area of memory in a record. In other words, one particular location in a record could be either of type A or of type B. This can be useful in either/or cases, where a record can have either one field or the other field, but not both. For instance, consider an office in which you track employees either

by their name or by an ID, but never by both at the same time. Here is a way to capture that idea in a variant `record`:

```
procedure TForm1.VariantButtonClick(Sender: TObject);
type
  TMyVariantRecord = record
    OfficeID: Integer;
    case PersonalID: Integer of
      0: (Name: ShortString);
      1: (NumericID: Integer);
  end;
var
  TomRecord, JaneRecord: TMyVariantRecord;
begin
  TomRecord.Name := 'Sammy';
  JaneRecord.NumericID := 42;
end;
```

The first field in this record, `OfficeID`, is just a normal field of the record. I've placed it there just so you can see that a variant record can contain not only a variant part, but also as many normal fields as you want.

The second part of the record, beginning with the word `case`, is the variant part. In this particular record, the `PersonalID` can be either a `String` or an `Integer`, but not both. To show how this works, I declare two records of type `TMyVariantRecord` and then use the `Name` part of one variant record and the `NumericID` of the other record instance.

`Name` and `NumericID` share the same block of memory in a `TVariantRecord`—or, more precisely, they both start at the same offset in the record. Needless to say, a `String` takes up more memory than an `Integer`. In any one instance of the record at any one point in time, you can have either the `String` or the `Integer`, but not both. Assigning a value to one or the other field, will, at least in principle, overwrite the memory in the other field.

The compiler will always allocate enough memory to hold the largest possible record that you can declare. For instance, in this case, it will always allocate 4 bytes for the field `OfficeID` and then 256 bytes for the `String`, giving you a total of 260 bytes for the record. This is the case even if you use only the `OfficeID` and `NumericID` fields, which take up only 8 bytes of memory.

For me, the most confusing part of variant records is the line that contains the word `case`. This syntax exists to give you a means of telling whether the record uses the `NumericID` or the `Name` field. Listing 3.4 shows a second example that focuses on how to use the `case` part of a variant record.

**LISTING 3.4** The `case` Part of a Variant Record Used in the `ShowRecordType` Method

```
unit Main;

interface

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

type
  TMyVariantRecord = record
    OfficeID: Integer;
    case PersonalID: Integer of
      0: (Name: ShortString);
      1: (NumericID: Integer);
  end;

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

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ShowRecordType(R: TMyVariantRecord);
begin
  case R.PersonalID of
    0: ShowMessage('Uses Name field: ' + R.Name);
    1: ShowMessage('Uses NumericID: ' + IntToStr(R.NumericID));
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
```

**LISTING 3.4** Continued

```
  Sam, Sue: TMyVariantRecord;
begin
  Sam.OfficeID := 1;
  Sue.PersonalID := 0;
  Sue.Name := 'Sue';
  ShowRecordType(Sue);
  Sam.OfficeID := 2;
  Sam.PersonalID := 1;
  Sam.NumericID := 1;
  ShowRecordType(Sam);
end;

end.
```

In this example, the two records are passed into the `ShowRecordType` method. One record has the `PersonalID` set to `0`; the other has it set to `1`. `ShowRecordType` uses this information to sort out whether any one particular record uses the `Name` field or the `RecordID` field:

```
case R.PersonalID of
  0: ShowMessage('Uses Name field: ' + R.Name);
  1: ShowMessage('Uses NumericID: ' + IntToStr(R.NumericID));
end;
```

As you can see, if `PersonalID` is set to `0`, the `Name` field is valid; if it is set to `1`, the `NumericID` field is valid.



**NOTE**

A `case` statement plays the same role in Object Pascal that `switch` statements play in C/C++ and Java. Other than the obvious syntactical similarities, there is no significant connection between the `case` part of a variant `record` and `case` statements. In and of themselves, `case` statements have nothing to do with variant `records`. It is simply coincidence that, in this example, `case` statements are the best way to sort out the two different types of records that can be passed to the `ShowRecordType` procedure. The following code, however, would also get the job done:

```
procedure TForm1.ShowRecordType2(R: TMyVariantRecord);
begin
  if R.PersonalID = 0 then
    ShowMessage('Uses Name field: ' + R.Name)
  else
    ShowMessage('Uses NumericID: ' + IntToStr(R.NumericID));
end;
```

Here is one last variant `record` that shows how far you can press this paradigm:

```
procedure TForm1.HonkerButtonClick(Sender: TObject);
type
  TAnimalType = (atCat, atDog, atPerson);
  TFurType = (ftWiry, ftSoft);
  TMoodType = (mtAggressive, mtCloying, mtGoodNatured);
  TAnimalRecord = record
    Name: String;
    Age: Integer;
    case AnimalType: TAnimalType of
      atCat: (Talk: ShortString;
            Fight: ShortString;
            Food: ShortString);
      atDog: (Fur: TFurType;
            Description: ShortString;
            Mood: TMoodType);
      atPerson:(Rank: ShortString;
             SerialNumber: ShortString);
  end;

var
  Cat, Dog, Person: TAnimalRecord;
begin
  Cat.Name := 'Valentine';
  Cat.Age := 14;
  Cat.Talk := 'Meow';
  Cat.Fight := 'Scratch';
  Cat.Food := 'Cheese';
  Dog.Name := 'Rover';
  Dog.Age := 2;
  Dog.AnimalType := atDog;
  Dog.Fur := ftWiry;
  Dog.Description := 'Barks';
  Dog.Mood := mtGoodNatured;
end;
```

In this example, I declare an enumerated type named `TAnimalType`. I use this type in the case part of the variant `record`, which enables me to use descriptive names rather than numbers for each case:

```
  TAnimalType = (atCat, atDog, atPerson);
  TAnimalRecord = record
    case AnimalType: TAnimalType of
      atCat:
      atDog:
      atPerson:
  end;
```

The other important bit of code in this example is the fact that you can place multiple fields in each element of the `case` part:

```
atDog: (Fur: TFurType;
        Description: ShortString;
        Mood: TMoodType);
```

Here you can see that the `atDog` part of the variant `records` has three separate fields: `Fur`, `Description`, and `Mood`.

**NOTE**

I use `ShortStrings` in these examples because you cannot use the system-allocated variables of type `AnsiStrings`, `WideStrings`, dynamic arrays, or `Interfaces` in a variant `record`. However, you can use pointers to these forbidden types.

Variant `records` don't play a large role in Object Pascal programming. However, they are used from time to time in some fairly crucial places, and they fit nicely into our search for the more advanced parts of the language that might not be obvious at first glance to a newcomer. In fact, I've seen many experienced Pascal programmers get tripped up by this syntactically complex entity.



# Pointers

Pascal pointers are just like C or C++ pointers: They are the key to what makes the language so powerful, but they can also cause you considerable pain.

I'll start by showing you what the untamed pointer type looks like in Object Pascal. Most people won't want to use pointers this way, but I think it helps to start by seeing the low-level usage. Then I'll show you the somewhat more civilized way pointers most often appear in Kylix programs. This is the undomesticated version:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  P: Pointer;
  S, Temp: String;
begin
  S := 'Sam';                // Set S = 'Sam'
  GetMem(P, 10);             // Allocate 10 bytes
  FillChar(P^, 10, 0);       // Zero out all 10 bytes
  Move(S[1], P^, Length(S)); // Move 'Sam' into P^
  SetLength(Temp, 4);        // Make Temp four bytes long
  Move(P^, Temp[1], 1);      // Move S from P to temp
```

```
  Inc(PChar(P));              // Make P point to am
  Move(P^, Temp[2], 1);       // Move a from P to temp
  Inc(PChar(P));              // Make P point to m
  Move(P^, Temp[3], 1);       // Move m from P to temp
  Temp[4] := #0;              // Set the null terminator for temp
  Dec(PChar(P), 2);           // Make P point at Sam
  FreeMem(P, 10);             // Free memory allocated for Sam
  Edit1.Text := Temp;
end;
```

This code is from a program named Pointers, which is in the Chap03 directory on your accompanying CD. The `Button1Click` method shown here does nothing practical other than show how to use pointer syntax in Object Pascal. In particular, the code creates a pointer of 10 bytes of memory. It zeroes out the bytes that the pointer addresses. Then it copies a string into the bytes address by the pointer. Finally, it accesses the contents of pointer, 1 byte at a time.

The Pascal pointer type is, logically enough, referred to as a pointer. Use the `^` operator operator>to address the memory associated with a pointer. The `^` operator performs more or less the same function in Pascal as the `->` operator does in C and C++.

I have commented this code so heavily that you can probably best follow it by simply reading the code itself. Notice, in particular, the following calls:

- The `GetMem` procedure allocates memory for a pointer. Pass the pointer in the first parameter and the number of bytes that you want to allocate in the second parameter. You will receive a copy of a fully initialized pointer to the amount of memory that you requested.
- `FillChar` can be used to spray a single-character value into multiple contiguous bytes in a very efficient manner. Commonly, it is used to zero out the bytes in a contiguous chunk of memory. In this case, I tell `FillChar` to fill the 10 bytes pointed to by `P` with the value `0`. This is the first character in the character set, not the number 0. The first parameter of the function specifies the memory that you want to fill, the second specifies the number of bytes of that memory that you want to address, and the third parameter is the character that you want to spray into those bytes. Needless to say, the routine is designed to be very fast.
- The `Move` procedure moves *n* number of bytes from one place in memory to another place. In this case, I move the 3 bytes addressed by the `String` variable into the bytes address by the pointer. The first parameter of the `Move` procedure is the source memory, the second is the destination memory, and the third is the number of bytes to be moved.
- Notice that I use the `Inc` and `Dec` procedures to iterate over particular bytes in the memory addressed by the pointer. This technique is known as pointer arithmetic, and it should be familiar to all C or C++ programmers. You will get a more detailed look at pointer

arithmetic in Chapter 5, in the sections on the debugger. After the call to `Move`, the pointer points at the letters `S`, `a`, and `m`. The last 7 bytes addressed by the pointer are still zeroed out. If I call `Inc(P)` one time, then the first byte addressed by the pointer is no longer `S`, but `a`. In C or C++, you would achieve the same effect by writing `P++`.

- The `FreeMem` procedure deallocates the memory assigned to the pointer `P`. Notice that I use `Dec` to unwind the memory addressed by the variable `P` back to the place initially allocated by the call to get `GetMem`. Failure to do this can yield undefined results.

The best way to understand these lines of code is to watch them very carefully in the debugger. Unfortunately, I have not yet talked to you about using the debugger. Furthermore, the parts of the debugger that you can use to really get a look at this code are relatively advanced features. As a result, I'm going to have to ask you to put this code aside for a time. Then, in Chapter 5, in the section "Using the Debugger," I will show you how to get a good look at what happens under the hood when this code is executed.

## Working with Pointers to Objects

The type of pointer that I showed you in the last section is really an advanced technique, not one that you are likely to employ in many sections of your code. Pointers to objects, however, are ubiquitous in Object Pascal, and you must understand them if you want to work with Kylix.



**NOTE**

In this section and the next, I talk a good deal about how objects work in Kylix. I assume that all readers of this book already understand objects as well as polymorphism, inheritance, encapsulation, virtual methods, and related topics. My goal in this text is not to explain OOP in and of itself, but instead to introduce or reintroduce Pascal objects to experienced programmers from other languages such as C++, VB, and Java.

Like `Strings`, pointers have a rather checkered history in Object Pascal. For many years, a Pascal pointer was a Pascal pointer, and as long as you understood how memory is addressed by a computer, they were fairly simple to use. In recent years, however, the waters have been muddied to make some changes that make programming easier for certain types of developers.

For many years, the basic rule for pointers was that if you wanted to address the memory they referenced, you used pointer syntax. For instance, you wrote `MyPointer^.SomeBlockOfMemory`. Spoken out loud, this is usually rendered as, "MyPointer points to `SomeBlockOfMemory`." This

syntax is easy for intermediate and advanced-level programmers to use. However, inexperienced programmers can find it disconcerting. To solve their feelings, the Pascal team decided to drop the `^` operator. The result is simple dot notation: `MyPointer.SomeBlockOfMemory`. When working with Pascal objects, you can now leave off the `^`.

Note, in particular, that all CLX objects should be addressed using dot notation. No one writes `TEdit^.Text`; they all write `TEdit.Text`.

You and I know, however, that this dropping of the pointer syntax is a simple affectation. It is a euphemism. Underneath this thin veneer, the untamed beast sharpens its claws on the bark of exotic tropical trees, its lidded eyes never still, its body crouching as it waits for you to make a false move. At this point it will—well, I'm not allowed to talk about that kind of thing in a book intended for general audiences. But if you have ever used a Microsoft product or attended a Bill Gate's demo, or if you have had a glimpse of the dreaded blue screen of death, then I'm sure you know something of what can happen.

**NOTE**

When they dropped the pointer syntax for objects, the Delphi and Kylix engineers also decided to place a `T` rather than a `P` before the declarations for these objects. For instance, it is proper to write `TEdit` or `TButton`, but not `PEdit` or `PButton`. Normally, you would put a `P` rather than a `T` before a pointer type. However, the game is properly afoot, and to play along you should use a `T` rather than a `P`.

The biggest issue when you use pointers is to make sure that the pointer is allocated properly and, even more importantly, that it is disposed of properly. In Kylix, these chores are largely taken out of the users' hands by two mechanisms:

1. In design mode, when you drop a component onto a form or onto another component, memory for that component is allocated automatically.
2. When a form closes, it automatically disposes of the memory associated with all the components that it owns. In fact, when any component that owns another component is destroyed, that component frees the memory for all the components that it owns. (In the section "Arrays", you saw the `Component` array that belongs to a form. We used that array for our purposes, but its primary purpose is to allow the form to track the components that it owns and to dispose of them properly when the time comes.)

Now you can begin to understand what the designers of Pascal have done:

- They've made Pascal allocate memory for you automatically most of the time when you're working in the Form Designer.

- When the time comes, Pascal will automatically dispose of the components.
- They've unburdened developers of the need to use the ^ syntax.

If you put these three pieces together, you can see that Delphi goes a long way toward entirely relieving you of the duty to think about pointers or to even know that pointers are being used.

All of that is well and good—except for one crucial fact: Object Pascal uses pointers everywhere! Every component that you drop on a form is a pointer to an object. In fact, it is not possible to create a static instance of an object in Pascal. All Pascal components must be created as pointers to components. Indeed, all Pascal objects must be instantiated as pointers!

Listing 3.5 shows how to explicitly create an instance of an object. This is information that you must understand for those times when you create objects on your own rather than by dropping them onto the Component Palette:

**LISTING 3.5** A Short Console Application Found on Disk as ObjectAllocate1.dpr

```
program ObjectAllocateOne;

{$APPTYPE CONSOLE}

uses SysUtils;

type
  TMyObject = class(TObject)
    procedure SayYourName;
  end;

{ MyObject }

procedure TMyObject.SayYourName;
begin
  WriteLn(ClassName);
end;

var
  MyObject: TMyObject;
begin
  MyObject := TMyObject.Create;
  MyObject.SayYourName;
  MyObject.Free;
end.
```

This program is a console application. You can create console applications by choosing File, New, Console Application from the menu system. Console applications are not run as part of X. Instead, they are run from the shell prompt and usually output only raw text.

**NOTE**

It is traditional to end console applications with the `ReadLn` statement while you are debugging them in the IDE. This ensures that the window that the program creates stays open until you press the Enter key. If you start Kylix from the shell prompt, there is no reason to use this mechanism. Instead, the text from your program will be seen at the shell prompt from which you started Kylix. (In fact, it might be the case in the first version of Kylix that this latter mechanism is the only way to see the output from your console applications.) Remember that when you start Kylix from the shell prompt, you should use the command `startkylix`.

The ObjectAllocateOne program declares a simple Object Pascal class named `TMyObject`. With rare exceptions, all Object Pascal classes should begin with `T`, which stands for "type." To declare the class, you write `TMyObject = class(TObject)`, where `TObject` is the name of the parent class.

To create an instance of `TMyObject`, you write the following code:

```
var
  MyObject: TMyObject;
begin
  MyObject := TMyObject.Create;
  ... // Code omitted here
```

This code allocates memory for an object. Java and C++ programmers would achieve the same ends by writing `MyObject = new TMyObject()`. VB programmers would write `Dim MyObject as TMyObject`. But parallels with VB and Java are inappropriate because Pascal treats pointers the same way C++ does. If you allocate memory for an object, you must free that memory or risk crippling your program!

Here is how to free memory for a Pascal object:

```
  MyObject.Free;
```

This code destroys the memory associated with `TMyObject`. After you have made this call, you can no longer safely reference `MyObject`.

## Pointers, Constructors, and Destructors

When you call `Free` on an object, the compiler also calls the destructor for your object, if there is one. The ObjectAllocateTwo program, shown in Listing 3.6, demonstrates how this works.

**LISTING 3.6** The ObjectAllocateTwo Program, Featuring a Constructor That Allocates Memory and a Destructor That Destroys That Same Memory

```
program ObjectAllocateTwo;
{$APPTYPE CONSOLE}
uses
  SysUtils;

type
  TMyObject = class(TObject)
  private
    FMyPointer: Pointer;
  protected
    constructor Create; virtual;
    destructor Destroy; override;
  public
    procedure SayYourName;
  end;

{ MyObject }

constructor TMyObject.Create;
begin
  inherited Create;
  GetMem(FMyPointer, 10);
end;

destructor TMyObject.Destroy;
begin
  FreeMem(FMyPointer, 10);
  inherited;
end;

procedure TMyObject.SayYourName;
begin
  WriteLn(ClassName);
end;

var
  MyObject: TMyObject;
```

**LISTING 3.6** Continued

```
begin
  MyObject := TMyObject.Create;
  MyObject.SayYourName;
  FreeAndNil(MyObject);
end.
```

Many programs contain pointers that need to be allocated and deallocated. When allocating memory for an object that will belong to another object, you should do one of two things:

- Declare the pointer or object as a method-scoped variable, and then allocate and deallocate the memory for that object inside one single method. This is the way the variable `P` is treated in the `Button1Click` method from the earlier section of this chapter titled "Pointers." In that method, `P` belongs to the `Button1Click` method, and the `Button1Click` method therefore takes on the chore of both allocating and deallocating memory for that object.
- The second method is shown in the AllocateObjectTwo program. In that program, the variable `FMyPointer` is declared as a field of `TMyObject`. (The name of all fields of Pascal objects should begin with `F`, which stands for "field.") The memory for `FMyPointer` is allocated in the object's constructor and deallocated in the object's destructor. If you do things in this way, you are unlikely to forget to either allocate or deallocate memory for your object. In particular, all you need do is see what pointers or objects are declared as fields of your object and then make sure that they are allocated in the constructor and deallocated in the destructor. If you have declared five fields for your object but only four of those fields are cleaned up in your destructor, you likely have a problem.

There is nothing in Object Pascal to keep you from allocating and deallocating memory for your objects and pointers wherever you please. These rules are merely guidelines. In some cases, it will not be possible to follow these guidelines, but you should heed them if you can.

## `virtual` Methods and the `override` Directive

In the AllocateObjectTwo program, the directive `virtual` is placed after the constructor for `TmyObject`, and the directive `override` follows the destructor. The first time you declare a method that you want to be virtual, use the directive `virtual`. If you are overriding an already existing `virtual` method, use the directive `override`.

All Pascal objects are guaranteed to have a destructor named `Destroy` and a method named `Free`. The purpose of the `Free` method is to call the destructor. More particularly, the `Free` method ensures that the object is not already set to `nil`; if it is not, the method calls `Destroy`.

Pascal objects inherit the `Free` and `Destroy` methods from `TObject`. When it comes time for a component to deallocate the memory for all the components that it owns, it simply calls `TObject.Free` on all the objects in its `Components` array. Through the miracle of polymorphism, this single call ensures that the destructors for each object are called. The end result is that all the memory for these objects is destroyed. This mechanism will not work, however, unless you override the `Destroy` method in your programs whenever you need to deallocate the memory associated with any of the fields of your object.

There is no need to override `Destroy` in the AllocateObjectOne program because the instance of `TMyObject` in that program does not have any fields for which it allocates memory. This is not the case in the AllocateObjectTwo program, so it is necessary to create a destructor for that object. Note further that in the AllocateObjectTwo program, `TMyObject` is not owned by a component. As a result, you must specifically call its destructor, as shown in that program. If `TMyObject` were owned by a component, you would not need to explicitly call `Free` on that object because its owner would perform the task for you.

**NOTE**

Newcomers to Object Pascal likely will have further questions at this point about the difference between objects and components. In general, a component is simply an object that you can place on the Component Palette. Saying as much, however, raises as many questions as it answers. Part II is dedicated to components, and there you will learn about them in depth.



I've tried to cover most of the major subjects about pointers that I think can cause confusion. If you want to hear more about pointers to objects, don't worry—the subject will come up again in Part II, when the focus of the text becomes CLX.

## What You Won't Find in Object Pascal

I want to close with a few words on what you won't find in Object Pascal:

- There is no support for multiple inheritance. However, you can descend from multiple interfaces, just as you can in Java.
- There is no support for operator overloading. This feature was deemed inappropriate for the language by its developers. They felt that it would make the syntax of the language too complex and possibly could lead to much longer compile times, similar to those found in C++. However, you will find that a clever use of variants will enable you to do something very similar to operating overloading.

- No garbage collection takes place. This is a feature that the team has often contemplated adding to the language. However, you can use numerous tricks to help you clean up memory automatically in Pascal; many of these were discussed in the section on pointers found earlier in this chapter.

## Summary

In this chapter, you learned about Object Pascal types. This information is essential if you want to use Object Pascal successfully.

The chapter began with a look at the simple types, such as integers, strings, and the floating-point types. Stepping into wilder terrain, you had a look at typecasting, arrays, and records. The last few pages of the text took you into the untamed jungle for a look at pointers and memory allocation for objects.

Hopefully this chapter has given you some sense of the power and subtlety of Object Pascal. The language is a subtle tool with many modern features. It gives you the speed and flexibility of C or C++, many of the modern programming features of Java, and the ease of use that you might expect from Perl or Python.

In the next chapter, you will learn more about working with objects and interfaces in an Object Pascal program. From there you will go on to learn more about the IDE and the editor. Finally, at the end of Part I, "Understanding Delphi and Linux," you will see how to use Object Pascal to access the core features of the X Window library.