Porting to Delphi 5

CHAPTER

15

IN THIS CHAPTER

If you're upgrading to Delphi 5 from a previous version, this chapter is written for you. The first section of this chapter discusses the issues involved in moving from any version of Delphi to Delphi 5. In the second, third, and fourth sections, you learn about the often subtle differences between the various 32-bit versions of Delphi and how to take these differences into account as you migrate applications to Delphi 5. The fourth section of this chapter is intended to help those migrating 16-bit Delphi 1.0 applications to the 32-bit world of Delphi 5. Although Borland makes a concerted effort to ensure that your code is compatible between versions, it's understandable that some changes have to be made in the name of progress, and certain situations require code changes if applications are to compile and run properly under the latest version of Delphi.

New to Delphi 5

In general, the more recent the version of Delphi you're coming from, the easier it will be for you to port to Delphi 5. However, whether you're migrating from Delphi 1, 2, 3, or 4, this section provides the information necessary for moving up to Delphi 5.

Which Version?

Although most Delphi code will compile for all versions of the compiler, in some instances language or VCL differences require that you write slightly differently to accomplish a given task for each product version. Occasionally, you might need to be able to compile for multiple versions of Delphi from one code base. For this purpose, each version of the Delphi compiler contains a `VER`*xxx* conditional define for which you can test in your source code. Because Borland C++Builder also ships with new versions of the Delphi compiler, these editions also contain this conditional define. Table 15.1 shows the conditional defines for the various versions of the Delphi compiler.

TABLE 15.1 Conditional Defines for Compiler Versions

Product	*Conditional Define*
Delphi 1	`VER80`
Delphi 2	`VER90`
C++Builder 1	`VER95`
Delphi 3	`VER100`
C++Builder 3	`VER110`
Delphi 4	`VER120`
C++Builder 4	`VER120`
Delphi 5	`VER130`

Using these defines, the source code you must write in order to compile for different compiler versions would look something like this:

```
{$IFDEF VER80}
  Delphi 1 code goes here
{$ENDIF}
{$IFDEF VER90}
  Delphi 2 code goes here
{$ENDIF}
{$IFDEF VER95}
  C++Builder 1 code goes here
{$ENDIF}
{$IFDEF VER100}
  Delphi 3 code goes here
{$ENDIF}
{$IFDEF VER110}
  C++Builder 3 code goes here
{$ENDIF}
{$IFDEF VER120}
  Delphi 4 and C++Builder 4 code goes here
{$ENDIF}
{$IFDEF VER130}
  Delphi 5 code goes here
{$ENDIF}
```

NOTE

If you're wondering why the Delphi 1.0 compiler is considered version 8, Delphi 2 version 9, and so on, it's because Delphi 1.0 is considered version 8 of Borland's Pascal compiler. The last Turbo Pascal version was 7.0, and Delphi is the evolution of that product line.

Just as you'll have to deal with differences in the language and VCL between Delphi versions, you'll also have to deal with differences in the Windows API. If you need to cope with differences in the 16-bit and 32-bit APIs from a single code base, you can take advantage of additional defines intended for this purpose. The 16-bit Delphi compiler defines `WINDOWS`, whereas the Win32 Delphi compilers define `WIN32`. The following code example demonstrates how to take advantage of these defines:

```
{$IFDEF WINDOWS}
  16-bit Windows-specific code goes here
{$ENDIF}
{$IFDEF WIN32}
  Win32-specific code goes here
{$ENDIF}
```

Units, Components, and Packages

Delphi 5 compiled units (DCU files) differ from those of all previous versions of Delphi (and C++Builder). You must have the source code to any units used in your application in order to build your application under any particular version of Delphi. This, of course, means that you won't be able to use any components used in your application—your own components or third-party components—unless you have the source to these components. If you don't have the source code to a particular third-party component, contact your vendor for a version of the component specific to your version of Delphi.

NOTE

This issue of compiler version versus unit file version is not a new situation and is the same as C++ compiler object file versioning. If you distribute (or buy) components without source code, you must understand that what you're distributing/buying is a compiler version–specific binary file that will probably need to be revised to keep up with subsequent compiler releases.

What's more, the issue of DCU versioning isn't necessarily a compiler-only issue. Even if the compiler weren't changed between versions, changes and enhancements to core VCL would probably still make it necessary that units be recompiled from source.

Delphi 3 also introduced *packages*, the idea of multiple units stored in a single binary file. Starting with Delphi 3, the component library became a collection of packages rather than one massive component library DLL. Like units, packages are not compatible across product versions, so you'll need to rebuild your packages for Delphi 5, and you'll need to contact the vendors of your third-party components for updated packages.

Migrating from Delphi 4

There are only a handful of migration issues as you take your Delphi 4 applications into Delphi 5. In many cases, you can simply load your project into Delphi 5 and hit the compile key. However, if you do run into problems, this section discusses the migration speed bumps you may face for getting things rolling in Delphi 5.

IDE Issues

Problems with IDE are likely the first you'll encounter as you migrate your applications. Here are a few of the issues you may encounter on the way:

- Delphi 4 debugger symbol files (RSM) are not compatible with Delphi 5. You'll know you're having this problem when you see the message "Error reading symbol file." If this happens, the fix is simple: Rebuild the application.

- Delphi 5 now defaults to storing form files in text mode. If you need to maintain DFM compatibility with earlier versions of Delphi, you'll need to save the forms files in binary instead. You can do this by unchecking New Forms As Text on the Preferences page of the Environment Options dialog.
- Code generation when importing and generating type libraries has been changed. In addition to some minor changes, the new generator has been enhanced to allow you to map symbol names; you can customize type library–to–Pascal symbol name mapping by editing the `tlibimp.sym` file. For directions, see the "Mapping Symbol Names in the Type Library" topic in the online help.

RTL Issues

The only issue you're likely to come across here deals with the setting of the floating-point unit (FPU) control word in DLLs. In previous versions of Delphi, DLLs would set the FPU control word, thereby changing the setting established by the host application. Now, DLL startup code no longer sets the FPU control word. If you need to set the control word to ensure some specific behavior by the FPU, you can do it manually using the `Set8087CW()` function in the `System` unit.

VCL Issues

There are a number of VCL issues that you may come across, but most involve some simple edits as a means to get your project on track in Delphi 5. Here's a list of these issues:

- The type of properties that represent an index into an image list has changed from `Integer` to `TImageIndex` type. `TImageIndex` is a strongly typed `Integer` defined in the `ImgList` unit as

  ```
  TImageIndex = type Integer;
  ```

  This should only cause problems in cases where exact type matching matters, such as when you're passing `var` parameters.
- `TCustomTreeview.CustomDrawItem()` has a new `var` parameter called `PaintImages` of type `Boolean`. If your application overrides this method, you'll need to add this parameter in order for it to compile in Delphi 5.
- The `CoInitFlags` variable in `ComObj`, which holds the flags passed to `CoInitializeEx()` in the `ComServ` unit, has been changed to properly support initialization of multithreaded COM servers. Now either the `COINIT_MULTITHREADED` or `COINIT_APARTMENTTHREADED` flags will be added when appropriate.
- If you're invoking pop-up menus in response to `WM_RBUTTONUP` messages or `OnMouseUp` events, you may exhibit "double" pop-up menus or no pop-up menus at all when compiling with Delphi 5. Delphi 5 now uses the `WM_CONTEXT` menu message to invoke pop-up menus.

Internet Development Issues

If you're developing applications with Internet support, we have some bad news and some good news:

- The `TWebBrowser` component, which encapsulates the Microsoft Internet Explorer ActiveX control, has replaced the `THTML` component from `Netmasters`. Although the `TWebBrowser` control is much more feature rich, you're faced with a good deal of rewrite if you used `THTML` because the interface is totally different. If you don't want to rewrite your code, you can go back to the old control by importing the `HTML.OCX` file from the `\Info\Extras\NetManage` directory on the Delphi 5 CD-ROM.
- Packages are now supported when building ISAPI and NSAPI DLLs. You can take advantage of this new support by replacing HTTPApp in your `uses` clause with WebBroker.

Database Issues

There are a few database issues that may trip you up as you migrate to Delphi 5. These involve some renaming of existing symbols and the new architecture of MIDAS:

- The type of the `TDatabase.OnLogin` event has been renamed `TDatabaseLoginEvent` from `TLoginEvent`. This is unlikely to cause problems, but you may run into troubles if you're creating and assigning to `OnLogin` in code.
- The global `FMTBCDToCurr()` and `CurrToFMTBCD()` routines have been replaced by the new `BCDToCurr` and `CurrToBCD` routines (and the corresponding protected methods on `TDataSet` have been replaced by the protected and undocumented `DataConvert` method).
- MIDAS has undergone some significant changes between Delphi 4 and 5. See Chapter 32, "MIDAS Development," for information on the changes, new features, and how to port your MIDAS applications to Delphi 5.

Migrating from Delphi 3

Although there aren't a great deal of compatibility issues between Delphi 3 and later versions, the few issues that do exist can be potentially more problematic than porting from any other previous version of Delphi to the next. Most of these issues revolve around new types and the changing behavior of certain existing types.

Unsigned 32-Bit Integers

Delphi 4 introduced the `LongWord` type, which is an unsigned 32-bit integer. In previous versions of Delphi, the largest integer type was a signed 32-bit integer. Because of this, many of

the types that you would expect to be unsigned, such as `DWORD`, `UINT`, `HResult`, `HWND`, `HINSTANCE`, and other handle types, were defined simply as `Integers`. In Delphi 4 and later, these types are redefined as `LongWords`. Additionally, the `Cardinal` type, which was previously a subrange type of `0..MaxInt`, is now also a `LongWord`. Although all this `LongWord` business won't cause problems in most circumstances, there are several problematic cases you should know about:

- `Integer` and `LongWord` are not `var`-parameter compatible. Therefore, you cannot pass a `LongWord` in a `var Integer` parameter, and vice versa. The compiler will give you an error in this case, so you'll need to change the parameter or variable type or typecast to get around this problem.
- Literal constants having the value of `$80000000` through `$FFFFFFFF` are considered `LongWords`. You must typecast such a literal to an `Integer` if you wish to assign it to an `Integer` type. Here's an example:

```
var
  I: Integer;
begin
  I := Integer($FFFFFFFF);
```

- Similarly, any literal having a negative value is out of range for a `LongWord`, and you'll need to typecast to assign a negative literal to a `LongWord`. Here's an example:

```
var
  L: LongWord;
begin
  L := LongWord(-1);
```

- If you mix signed and unsigned integers in arithmetic or comparison operations, the compiler will automatically promote each operand to `Int64` in order to perform the arithmetic or comparison. This can cause some very difficult-to-find bugs. Consider the following code:

```
var
  I: Integer;
  D: DWORD;
begin
  I := -1;
  D := $FFFFFFFF;
  if I = D then DoSomething;
```

  Under Delphi 3, *`DoSomething`* would execute because `-1` and `$FFFFFFFF` are the same value when contained in an `Integer`. However, because Delphi 4 and later will promote each operand to `Int64` in order to perform the most accurate comparison, the generated code ends up comparing `$FFFFFFFFFFFFFFFF` against `$00000000FFFFFFFF`, which is definitely not what's intended. In this case, *`DoSomething`* will not execute.

TIP

The compiler in Delphi 4 and later generates a number of new hints, warnings, and errors that deal with these type compatibility problems and implicit type promotions. Make sure you turn on hints and warnings when compiling in order to let the compiler help you write clean code.

64-Bit Integer

Delphi 4 also introduced a new type called `Int64`, which is a signed 64-bit integer. This new type is now used in the RTL and VCL where appropriate. For example, the `Trunc()` and `Round()` standard functions now return `Int64`, and there are new versions of `IntToStr()`, `IntToHex()`, and related functions that deal with `Int64`.

The Real Type

Starting with Delphi 4, the `Real` type became an alias for the `Double` type. In previous versions of Delphi and Turbo Pascal, `Real` was a six-byte floating-point type. This shouldn't pose any problems for your code unless you have `Real`s written to some external storage (such as a `file of record`) with an earlier version or you have code that depends on the organization of the `Real` in memory. You can force `Real` to be the old 6-byte type by including the `{$REALCOMPATIBILITY ON}` directive in the units you want to use the old behavior. If all you need to do is force a limited number of instances of the `Real` type to use the old behavior, you can use the `Real48` type instead.

Migrating from Delphi 2

You'll find that a high degree of compatibility between Delphi 2 and the later versions means a smooth transition into a more up-to-date Delphi version. However, some changes have been made since Delphi 2, both in the language and in VCL, that you'll need to be aware of to migrate to the latest version and take full advantage of its power.

Changes to Boolean Types

The implementation of the Delphi 2 Boolean types (`Boolean`, `ByteBool`, `WordBool`, `LongBool`) dictated that `True` was ordinal value `1` and `False` ordinal value `0`. To provide better compatibility with the Win32 API, the implementations of `ByteBool`, `WordBool`, and `LongBool` have changed slightly; the ordinal value of `True` is now `-1` (`$FF`, `$FFFF`, and `$FFFFFFFF`, respectively). Note that no change was made to the `Boolean` type. These changes have the potential to cause problems in your code—but only if you depend on the ordinal values of these types. For example, consider the following declaration:

```
var
  A: array[LongBool] of Integer;
```

This code is quite harmless under Delphi 2; it declares an `array[False..True]` (or `[0..1]`) `of Integer`, for a total of three elements. Under Delphi 3 and later, however, this declaration can cause some very unexpected results. Because `True` is defined as `$FFFFFFFF` for a `LongBool`, the declaration boils down to `array[0..$FFFFFFFF] of Integer`, or an array of 4 billion `Integers`! To avoid this problem, use the `Boolean` type as the array index.

Ironically, this change was necessary because a disturbing number of ActiveX controls and control containers (such Visual Basic) test `BOOLs` by checking for `-1` rather than testing for a zero or nonzero value.

TIP

To help ensure portability and to avoid bugs, never write code like this:

```
if BoolVar = True then ...
```

Instead, always test Boolean types like this:

```
if BoolVar then ...
```

ResourceString

If your application uses string resources, consider taking advantage of `ResourceStrings` as described in Chapter 2, "The Object Pascal Language." Although this won't improve the efficiency of your application in terms of size or speed, it will make language translation easier. `ResourceStrings` and the related topic of resource DLLs are required to be able to write applications displaying different language strings but have them all running on the same core VCL package.

RTL Changes

Several changes made to the runtime library (RTL) after Delphi 2 might cause problems as you migrate your applications. First, the meaning of the `HInstance` global variable has changed slightly: `HInstance` contains the instance handle of the current DLL, EXE, or package. Use the new `MainInstance` global variable when you want to obtain the instance handle of the main application.

The second significant change pertains to the `IsLibrary` global. In Delphi 2, you could check the value of `IsLibrary` to determine whether your code was executing within the context of a DLL or EXE. `IsLibrary` isn't package aware, however, so you can no longer depend on `IsLibrary` to be accurate, depending on whether it's called from an EXE, DLL, or a module

within a package. Instead, you should use the `ModuleIsLib` global, which returns `True` when called within the context of a DLL or package. You can use this in combination with the `ModuleIsPackage` global to distinguish between a DLL and a package.

TCustomForm

The Delphi 3 VCL introduced a new class between `TScrollingWinControl` and `TForm` called `TCustomForm`. In itself, that shouldn't pose a problem for you in migrating your applications from Delphi 2; however, if you have any code that manipulates instances of `TForm`, you might need to update it so that it manipulates `TCustomForms` instead of `TForms`. Some examples of these are calls to `GetParentForm()`, `ValidParentForm()`, and any usage of the `TDesigner` class.

CAUTION

The semantics for `GetParentForm()`, `ValidParentForm()`, and other VCL methods that return `Parent` pointers have changed slightly from Delphi 2. These routines can now return `nil`, even though your component has a parent window context in which to draw. For example, when your component is encapsulated as an ActiveX control, it may have a `ParentWindow`, but not a `Parent` control. This means you must watch out for Delphi 2 code that does this:

```
with GetParentForm(xx) do ...
```

`GetParentForm()` can now return `nil` depending on how your component is being contained.

GetChildren()

Component writers, be aware that the declaration of `TComponent.GetChildren()` has changed to read as follows:

```
procedure GetChildren(Proc: TGetChildProc; Root: TComponent); dynamic;
```

The new `Root` parameter holds the component's root owner—that is, the component obtained by walking up the chain of the component's owners until `Owner` is `nil`.

Automation Servers

The code required for automation has changed significantly from Delphi 2. Chapter 23, "COM-based Technologies," describes the process of creating Automation servers in Delphi 5. Rather than describe the details of the differences here, suffice it to say that you should never mix the Delphi 2 style of creating Automation servers with the more recent style found in Delphi 3 and later.

In Delphi 2, automation is facilitated through the infrastructure provided in the `OleAuto` and `Ole2` units. These units are present in later releases of Delphi only for backward compatibility, and you shouldn't use them for new projects. Now the same functionality is provided in the `ComObj`, `ComServ`, and `ActiveX` units. You should never mix the former units with the latter in the same project.

Migrating from Delphi 1

Most of the changes required when porting Delphi 1 applications to a later version are necessary because of the nature of programming under a new operating system. Other changes are required because of enhancements in VCL and the Object Pascal language. Some people would prefer that Delphi 1 applications simply ran without modification under Delphi 5. If that's your opinion, keep in mind that sometimes it's necessary to leave a little behind to move ahead. You should find that each new version of Delphi strikes an excellent balance in this regard.

In addition to explaining what you need to know to migrate Delphi 1 applications, this section also provides you with information about optimizing your project for 32-bit Delphi and maintaining code that's compatible with both 16- and 32-bit Delphi.

Strings and Characters

In response to customer demand for a more flexible string, Borland introduced a new string type in Delphi 2 known as `AnsiString`. Among other benefits, `AnsiString` supports the creation of virtually unlimited-length strings. Delphi 2 also introduced new character and null-terminated string types to fully support application internationalization using the Unicode double-byte format. Delphi 3 took double-byte support even further with the introduction of the `WideString` type. By far the most common issues likely to arise as you migrate from Delphi 1 are those dealing with the use and manipulation of strings.

New Character Types

Strings, of course, are made up of characters, so it's important that you understand the behavior of character types in Delphi 4 before learning about the new string types. The most important change to this portion of the language is the new character type `WideChar`, introduced in Delphi 2 to support Unicode (or "wide") character and string types. In addition to `WideChar`, Delphi 3 introduced a new type name, `AnsiChar`, which specifies a normal single-byte character.

The `AnsiChar` type is the same as the Delphi 1 `Char` type. It's a one-byte value that can contain any of 256 different values. Use `AnsiChar` only when you know that the value in question will always be one byte in size.

Use `WideChar` for a character value that's two bytes in size. `WideChar` exists for compatibility with the Unicode character standard adopted by the Win32 API to support local-language

strings. Because of its two-byte size, a `WideChar` can contain any one of 65,536 possible values, enough for even the largest alphabets.

> The purpose of Unicode is to support use of local-language strings across the entire system (and across the global network) without loss of information. There are one-byte character sets for most languages (except Far Eastern languages), but converting between language character sets is not always reversible, so conversion introduces loss of information. Unicode solves that by eliminating the need to convert between character set encodings. It also addresses the Far Eastern language issue by providing a very large character set base for encoding pictograph-per-word languages such as Chinese.

Of course, the `char` type is still valid in Delphi. Currently, the `char` and `AnsiChar` types are equivalent. However, Borland reserves the right to change the definition of `char` to `WideChar` in some future version of Delphi; you should never depend on the size of a `char` being a certain length in your code—always use `SizeOf()` to determine its actual size.

New String Types

Listed here are the string types supported in Delphi 5:

- `AnsiString` (also referred to as *long string* and *huge string*) is the new default string type for Object Pascal. It's composed of `AnsiChar` characters and allows for lengths of up to 1GB. This string type is also compatible with null-terminated strings. This string is always dynamically allocated and lifetime managed.
- `ShortString` is synonymous with the standard string type in Delphi 1.0. The capacity of `ShortString` is limited to 255 characters.
- `WideString` is comprised of `WideChar` characters, and, like `AnsiString`, it's automatically allocated and lifetime managed. Chapter 2, "The Object Pascal Language," contains a complete rundown on this and other string types.
- `PAnsiChar` is a pointer to a null-terminated `AnsiChar` string.
- `PWideChar` is a pointer to a null-terminated string of `WideChar` characters, making up a Unicode, or double-byte, string.
- `PChar` is a pointer to a null-terminated `Char` string, which is fully compatible with C-style strings used in Windows API functions. This type hasn't changed from version 1.0 and is currently defined as `PAnsiChar`.

By default, strings defined in Delphi 2 and later are `AnsiStrings`. So if you declare a `string`, as shown here, the compiler assumes that you're creating an `AnsiString`:

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

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

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

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

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

CAUTION

Be careful when passing strings declared in units with `$H+` to functions and procedures defined in units with `$H-`, and vice versa. These types of errors can introduce some hard-to-find bugs into your applications.

Setting String Length

In Delphi 1, you could set the length of a string by assigning a value to the 0, byte as shown here:

```
S[0] := 23;   { sets the length byte of a short string }
```

This was possible because the maximum length of a short string (255) could be stored in the leading byte. Because the maximum length of a long string is 1GB, the size obviously can't fit in one byte; the length is therefore stored differently. Because of this issue, Delphi 2 introduced a new standard procedure called `SetLength()` that you should use to set the length of a string. `SetLength()` is defined as follows:

```
procedure SetLength(var S: String; NewLength: Integer};
```

`SetLength()` can be used with short and long strings. If you want to maintain one set of source code for 16-bit Delphi 1 projects and 32-bit Delphi projects, you can define a `SetLength()` function, as follows, for 16-bit Delphi 1 projects:

```
{$IFDEF WINDOWS}
{ for 16-bit Delphi 1 projects }
```

```
procedure SetLength(var S: String; NewLength: Integer);
begin
  S[0] := Char(NewLength);
end;
{$ENDIF}
```

TIP

For more information on the physical layout of the `AnsiString` type, see Chapter 2, "The Object Pascal Language."

Dynamically Allocated Strings

In Delphi 1, it's possible to use variables of type `PString` to implement dynamically allocated strings by allocating memory with the `NewStr()`, `GetMem()`, or `AllocMem()` standard procedure. In 32-bit Delphi, because long strings are automatically allocated dynamically from the heap, there's no need to use such techniques. Change your `PString` references to `String` (the code that dynamically creates and frees memory). You must also remove any dereferencing of the `PString` variable that appears in your code. Consider the following block of Delphi 1 code:

```
var
  S1, S2: PString;
begin
  S1 := AllocMem(SizeOf(S1^));
  S1^ := 'Give up the rock.';
  S2 := NewStr(S1^);
  FreeMem(S1, SizeOf(S1^));
  Edit1.Text := S2^;
  DisposeStr(S2);
end;
```

This code can be enormously simplified (and optimized) simply by taking advantage of long strings, as shown here:

```
var
  S1, S2: string;
begin
  S1 := 'Give up the rock.';
  S2 := S1;
  Edit1.Text := S2;
end;
```

Indexing Strings as Arrays

Sometimes you want to access a certain character in a string by indexing the string as an array. For example, the following line of code sets the fifth character in the string to *A*:

```
S[5] := 'A';
```

This type of operation is still perfectly legitimate with long strings, but there's one caveat: Because long strings are dynamically allocated, you must ensure that the length of the string is greater than or equal to the character element you attempt to index. For example, the following code is invalid:

```
var
  S: string;
begin
  S[5] := 'A';  // Space for S has not yet been allocated!!
end;
```

However, this code is quite valid:

```
var
  S: string;
begin
  S := 'Hello';  // allocates enough room for the string
  S[5] := 'A';
end;
```

This code is also valid:

```
var
  S: string;
begin
  SetLength(S, 5); // allocate 5 characters for S
  S[5] := 'A';
end;
```

CAUTION

You should not assume that a character index into a string is the same thing as the byte offset into the string. For example, `WideStringvVar[5]` accesses the fifth character (at byte offset 10).

Null-Terminated Strings

When calling Windows 3.1 API functions in Delphi 1, programmers had to be aware of the difference between the Pascal `String` type and the C-style `PChar` (the null-terminated string used in Windows). Long strings make it much easier to call Win32 API functions. Long strings are both heap allocated and guaranteed to be null terminated. For these reasons, you can simply typecast a long string variable when you need to use it as a null-terminated `PChar` in a Win32 API function call. Imagine that you have procedure `Foo()`, defined as follows:

```
procedure Foo(P: PChar);
```

In Delphi 1, you would typically call this function like this:

```
var
  S: string;            { Pascal short string }
  P: PChar;             { null terminated string }
begin
  S := 'Hello world';   { initialize S }
  P := AllocMem(255);   { allocate P }
  StrPCopy(P, S);       { copy S to P }
  Foo(P);               { call Foo with P }
  FreeMem(P, 255);      { dispose P }
end;
```

Using a 32-bit version of Delphi, you can call `Foo()` using a long string variable with the following syntax:

```
var
  S: string;       // a long string is null terminated
begin
  S := 'Hello World';
  Foo(PChar(S));   // fully compatible with PChar type
end;
```

This means that you can optimize your 32-bit code by removing unnecessary temporary buffers to hold null-terminated strings.

Null-Terminated Strings as Buffers

A common use for `PChar` variables is as a buffer to be passed to an API function that fills the buffer string with information. A classic example is the `GetWindowsDirectory()` API function, defined in the Win32 API as follows:

```
function GetWindowsDirectory(lpBuffer: PChar; uSize: UINT): UINT;
```

If your goal is to store the Windows directory in a string variable, a common shortcut under Delphi 1 is to pass the address of the first element of the string as shown here:

```
var
  S: string;
begin
  GetWindowsDirectory(@S[1], 254); { 254 = room for null }
  S[0] := Chr(StrLen(@S[1]));      { adjust length }
end;
```

This technique doesn't work with long strings for two reasons. First, as mentioned earlier, you must give the string an initial length before any space is allocated. Second, because a long

string is already a pointer to heap space, using the `@` operator effectively passes a pointer to a pointer to a character—definitely not what you intended! With long strings, this technique is streamlined by typecasting the string to a `PChar`:

```
var
  S: string;
begin
  SetLength(S, MAX_PATH + 1);      // allocate space
  GetWindowsDirectory(PChar(S), MAX_PATH);
  SetLength(S, StrLen(PChar(S))); // adjust length
end;
```

PChars as Strings

Because long strings can be used as `PChars`, it's only fair that the reverse hold true. Null-terminated strings are assignment compatible to long strings. In Delphi 1, the following code requires a call to `StrPCopy()`:

```
var
  S: string;
  P: PChar;
begin
  P := StrNew('Object Pascal');
  S := StrPas(P);
  StrDispose(P);
end;
```

Now you can accomplish the same thing with a simple assignment using long strings:

```
var
  S: string;
  P: PChar;
begin
  P := StrNew('Object Pascal');
  S := P;
  StrDispose(P);
end;
```

Similarly, you can also pass null-terminated strings to functions and procedures that expect `String` parameters. Suppose that procedure `Bar()` is defined as follows:

```
procedure Bar(S: string);
```

You can call `Bar()` using a `PChar` as follows:

```
var
  P: PChar;
begin
  P := StrNew('Hello');
```

```
  Bar(P);
  StrDispose(P);
end;
```

However, this technique doesn't work with procedures and functions that accept `String`s by reference. Suppose that procedure `Bar()` were instead defined like this:

```
procedure Bar(var S: string);
```

The sample code just presented couldn't be used to call `Bar()`. Instead, you have to define a temporary string to pass to `Bar()`, as shown here:

```
var
  P: PChar;
  TempStr: string;
begin
  P := StrNew('Hello');
  TempStr := P;
  Bar(TempStr);
  StrDispose(P);
  P := PChar(TempStr);
end;
```

Delphi 2 introduced a standard procedure called `SetString()` that allows you to copy only a portion of a `PChar` into a string variable. `SetString()` has an advantage in that it works with both long and short strings. The definition of `SetString()` is given here:

```
procedure SetString(var S: string: Buffer: PChar; Len: Integer);
```

CAUTION

Be wary of assigning a string variable to a `PChar` variable when the lifetime of the `PChar` variable is greater than that of the string. Because the string will be deallocated when it leaves scope, the `PChar` variable will point to garbage after the string leaves scope. The following code illustrates this problem:

```
var
  P: PChar;

procedure Bar(var P: PChar);
var
  S: String;
begin
  S := 'Hola Mundo';
  P := PChar(S);   // P is valid here
end;               // S is freed here
```

```
procedure Foo;
begin
  Bar(P);
  ShowMessage(P);  // DANGER! P is now invalid
end;
```

Variable Size and Range

Another issue that might arise as you migrate your Delphi code is that some types change size (and therefore range) when they move from 16-bit to 32-bit environments. Tables 15.2 and 15.3 show the differences with regard to these types.

TABLE 15.2 Variable Size Differences

Type	*16-Bit Size*	*32-Bit Size*
`Integer`	Two bytes	Four bytes
`Cardinal`	Two bytes	Four bytes
`String`	256 bytes	Four bytes

TABLE 15.3 Variable Range Differences

Type	*16-Bit Range*	*32-Bit Range*
`Integer`	-32,768..32,767	-2,147,483,648..2,147,483,647
`Cardinal`	0..65,536	0..2,147,483,647
`String`	255 characters	1GB of characters

For the most part, these new variable sizes have no effect on your applications. In those areas where you depend on type sizes, make sure you use the `SizeOf()` function. Also, if you've written any of these types to binary files or BLOBs in 16-bit Delphi, you must take into account the change in size as you read the data back in with 32-bit Delphi. For this purpose, Table 15.4 indicates which 32-bit Delphi types share binary compatibility with the Delphi 1 types.

TABLE 15.4 Variable Type Compatibility

16-Bit Delphi Type	*Compatible 32-Bit Delphi Type*
`Integer`	`SmallInt`
`Cardinal`	`Word`
`string`	`ShortString`

Record Alignment

By default in 32-bit Delphi, records are padded so that they're aligned properly; 32-bit data (such as `Integer`) is aligned on 32-bit (`DWORD`) boundaries, and 16-bit data (such as `Word`) is aligned on addresses that are even multiples of 16.

```
type
  TX = record
    B: Byte;
    L: Longint;
  end;
```

NOTE

Data is aligned on boundaries in order to optimize processor performance when accessing memory.

With the default compiler settings, the Delphi 1 `SizeOf(TX)` function returns `5`; under Delphi 2 and later, `SizeOf(TX)` function returns `8`. This isn't normally an issue; however, it can be an issue if you don't use `SizeOf()` to determine the size of the record in your code or if you have records written to a binary file.

The reason that the 32-bit Delphi compiler aligns record elements on `DWORD` boundaries is that doing so enables the compiler to generate more optimized code. If there's a reason you want to block this behavior, you can use the new `packed` modifier in the `type` declaration:

```
type
  TX = packed record
    B: Byte;
    L: Longint;
  end;
```

With `TX` defined as a *packed record*, as shown here, `SizeOf(TX)` now returns `5` under Delphi 2 and later. You can make packed records the default by using the `$A-` compiler directive.

TIP

If you can reorder the fields in a record to make fields start on their natural boundaries (for example, reorder four `Byte` fields so they all come before an `Integer` field, instead of two on either side of the `Integer`), you can get optimum packing without incurring the performance cost of unaligned data.

32-Bit Math

A much more subtle issue regarding variable size is that the 32-bit Delphi compiler automatically performs optimized 32-bit math on all operands in an expression (Delphi 1 used 16-bit math). Consider the following Object Pascal code:

```
var
  L: longint;
  w1, w2: word;
begin
  w1 := $FFFE;
  w2 := 5;
  L := w1 + w2;
end;
```

Under Delphi 1.0, the value of `L` at the end of this routine is `3` because the calculation of `w1 + w2` is stored as a 16-bit value, and the operation causes the result to wrap. Under Delphi 3, the value of `L` at the end of this routine is `$10003` because the `w1 + w2` calculation is performed using 32-bit math. The repercussion of the new functionality is that if you use and depend on the compiler's range-checking logic to catch "errors" such as these in Delphi 1.0, you must use some other method for finding those errors in 32-bit versions of Delphi, because a range-check error won't occur.

The TDateTime Type

To maintain compatibility with OLE and the Win32 API, the zero value of a `TDateTime` variable has changed. Date values start at `00/00/0000` under Delphi 1; they start at `12/30/1899` under 32-bit Delphi. Although this change won't affect dates stored in a database field, it will affect binary dates stored in a binary file or database BLOB field.

Unit Finalization

Delphi 1 provides a procedure called `AddExitProc()` and a pointer called `ExitProc` that enable you to define a procedure as containing "exit code" for a particular unit. Under 32-bit Delphi, the process of adding an exit procedure to a unit is greatly simplified with the addition of the unit's `finalization` section. Intended as a counterpart to the unit's `initialization` section, the code in a `finalization` section is guaranteed to be called when the application closes. Although this type of change isn't necessary to compile your application under 32-bit Delphi, it does make for much cleaner code.

NOTE

Conversion of `ExitProc`s to `finalization` blocks is mandatory for packages. Packages can be dynamically loaded and unloaded multiple times at design time, and `ExitProc`s are not called when a package is dynamically unloaded by the IDE. Therefore, your cleanup code must go in `finalization` sections.

Consider the following Delphi 1 `initialization` section and exit code:

```
procedure MyExitProc;
begin
  MyGlobalObject.Free;
end;
initialization
  AddExitProc(MyExitProc);
  MyGlobalObject := TGlobalObject.Create;
end.
```

This code can be simplified using the `finalization` section in 32-bit Delphi, as shown here:

```
initialization
  MyGlobalObject := TGlobalObject.Create;
finalization
  MyGlobalObject.Free;
end.
```

Assembly Language

Because assembly language is highly dependent on the platform for which it is written, the 16-bit built-in assembly language in Delphi 1 applications doesn't work in 32-bit Delphi. You must rewrite such routines using 32-bit assembly language.

Additionally, certain interrupts might not be supported under Win32. An example of interrupts no longer supported under Win32 is the suite of DOS Protected Mode Interface (DPMI) functions provided under interrupt `$31`. In some cases, Win32 API functions and procedures take the place of interrupts (the new Win32 file I/O functions are an example). If your application makes use of interrupts, refer to the Win32 documentation to check the alternatives in your specific case.

Additionally, inline hexadecimal code is no longer supported in the 32-bit Delphi compiler. If you have any routines that use inline code, replace them with 32-bit assembly language routines.

Calling Conventions

Delphi 1 can use either the `cdecl` or `pascal` calling convention for parameter passing and stack cleanup for function and procedure calls. The default calling convention for Delphi 1 is `pascal`.

Delphi 2 introduced directives representing two new calling conventions: `register` and `stdcall`. The `register` calling convention is the default for Delphi 2 and 3, offering faster performance. This method dictates that the first three 32-bit parameters be passed in the `eax`, `edx`, and `ecx` registers, respectively. Remaining parameters use the `pascal` calling convention. The `stdcall` calling convention is a hybrid of `pascal` and `cdecl` in that the parameters are passed using the `cdecl` convention but the stack is cleaned up using the `pascal` convention.

Delphi 3 introduced a new procedure directive called `safecall`. `safecall` follows the `stdcall` convention for parameter passing and also allows COM errors to be handled in a more Delphi-like manner. Most COM functions return `HRESULT` values as errors, whereas the preferred manner of error handling in Delphi is through the use of structured exception handling. When you call a `safecall` function from Delphi, the `HRESULT` return value of the function is converted into an exception that you may handle. When implementing a `safecall` function in Delphi, any exceptions raised in the function will be converted into an `HRESULT` value, which is returned to the caller.

NOTE

Although functions and procedures in the 16-bit Windows API use the `pascal` calling convention, Win32 API functions and procedures use the `stdcall` convention. Consequently, if you have any callback functions in your code, those also use the `stdcall` calling convention. Consider the following callback, intended for use with the `EnumWindows()` API function under 16-bit Windows:

```
function EnumWindowsProc(Handle: hwnd; lParam: Longint): BOOL; export;
```

It's defined as follows for 32-bit Windows:

```
function EnumWindowsProc(Handle: hwnd; lParam: Longint): BOOL; stdcall;
```

Dynamic Link Libraries (DLLs)

The creation and use of DLLs work very much the same in 32-bit Delphi as in Delphi 1, although there are a few minor differences. Some of these issues are listed here:

- Because of Win32's flat memory model, the `export` directive (necessary for callback and DLL functions in Delphi 1) is unnecessary in later versions. It's simply ignored by the compiler.

- If you're writing a DLL you intend to share with executables written using other development tools, it's a good idea to use the `stdcall` directive for maximum compatibility.
- The preferred way to export functions in a Win32 DLL is by name (instead of by ordinal). The following example exports functions by ordinal, the Delphi 1 way:

```
function SomeFunction: integer; export;
begin
.
.
.
end;

procedure SomeProcedure; export;
begin
.
.
.
end;

exports
  SomeFunction index 1,
  SomeProcedure index 2;
```

  Here are the same functions exported by name, the 32-bit Delphi way:

```
function SomeFunction: integer; stdcall;
begin
.
.
.
end;

procedure SomeProcedure; stdcall;
begin
.
.
.
end;

exports
  SomeFunction name 'SomeFunction';
  SomeProcedure name 'SomeProcedure';
```

- Exported names are case sensitive. You must use proper case when importing functions by name and when calling `GetProcAddress()`.
- When you import a function or procedure and specify the library name after the external directive, the file extension can be included. If no extension is specified, `.DLL` is assumed.

- Under Windows 3.*x*, a DLL in memory has only one data segment that's shared by all instances of the DLL. Therefore, if applications A and B both load DLL C, changes made to global variables in DLL C from application A are visible to application B, and the reverse is also true. Under Win32, each DLL receives its own data segment, so changes made to global DLL data from one program aren't visible to another program.

TIP

See Chapter 9, "Dynamic Link Libraries," for more information on the behavior of DLLs under Win32.

Windows Operating System Changes

In several areas, changes in the 32-bit architecture of Windows can have an impact on code written in Delphi. These include changes resulting from the 32-bit memory model, changes in resource formats, unsupported features, and changes to the Windows API itself.

32-Bit Address Space

Win32 provides a 4GB flat address space for your application. The term *flat* means that all segment registers hold the same value and that the definition of a pointer is an offset into that 4GB space. Because of this, any code in your Delphi 1 applications that depends on the concept of a pointer consisting of a selector and offset must be rewritten to accommodate the new architecture.

The following elements of the Delphi 1 runtime library are 16-bit pointer specific and are not in the 32-bit Delphi runtime library: `DSeg`, `SSeg`, `CSeg`, `Seg`, `Ofs`, and `SPtr`.

Because of the way in which Win32 uses a hard disk paging file to simulate RAM on demand, the Delphi 1.0 `MemAvail()` and `MaxAvail()` functions are no longer useful for gauging available memory. If you have to obtain this information in 32-bit Delphi, use the `GetHeapStatus()` Delphi RTL function, which is defined as follows:

```
function GetHeapStatus: THeapStatus;
```

The `THeapStatus` record is designed to provide information (in bytes) on the status of the heap for your process. This record is defined as follows:

```
type
  THeapStatus = record
    TotalAddrSpace: Cardinal;
    TotalUncommitted: Cardinal;
    TotalCommitted: Cardinal;
    TotalAllocated: Cardinal;
```

```
    TotalFree: Cardinal;
    FreeSmall: Cardinal;
    FreeBig: Cardinal;
    Unused: Cardinal;
    Overhead: Cardinal;
    HeapErrorCode: Cardinal;
  end;
```

Again, because the nature of Win32 is such that the amount of "free" memory has little meaning, most users will find the `TotalAllocated` field (which indicates how much heap memory has been allocated by the current process) most useful for debugging purposes.

NOTE

For more information on the internals of the Win32 operating system, see Chapter 3, "The Win32 API."

32-Bit Resources

If you have any resources (RES or DCR files) that you link into your application or use with a component, you must create 32-bit versions of these files before you can use them with 32-bit Delphi. Typically, this is a simple matter of using the included Image Editor or a separate resource editor (such as Resource Workshop) to save the resource file in a 32-bit-compatible format.

VBX Controls

Because Microsoft doesn't support VBX controls (which are inherently 16-bit controls) in 32-bit applications for Windows 95 and Windows NT, they aren't supported in 32-bit Delphi. ActiveX controls (OCXs) effectively replace VBX controls in 32-bit platforms. If you want to migrate a Delphi 1.0 application that uses VBX controls, contact your VBX vendor to get an equivalent 32-bit ActiveX control.

Changes to the Windows API Functions

Some Windows APIs or features have changed from Windows 3.1 to Win32. Some 16-bit API functions no longer exist in Win32, some functions are obsolete but continue to exist for the sake of compatibility, and some accept different parameters or return different types or values. Tables 15.5 and 15.6 list these functions. For complete documentation on these functions, see the Win32 API online help that comes with Delphi.

TABLE 15.5 Obsolete Windows 3.*x* API Functions

Windows 3.x Function	*Win32 Replacement*
`OpenComm()`	`CreateFile()`
`CloseComm()`	`CloseHandle()`
`FlushComm()`	`PurgeComm()`
`GetCommError()`	`ClearCommError()`
`ReadComm()`	`ReadFile()`
`WriteComm()`	`WriteFile()`
`UngetCommChar()`	N/A
`DlgDirSelect()`	`DlgDirSelectEx()`
`DlgDirSelectComboBox()`	`DlgDirSelectComboBoxEx()`
`GetBitmapDimension()`	`GetBitmapDimensionEx()`
`SetBitmapDimension()`	`SetBitmapDimensionEx()`
`GetBrushOrg()`	`GetBrushOrgEx()`
`GetAspectRatioFilter()`	`GetAspectRatioFilterEx()`
`GetTextExtent()`	`GetTextExtentPoint()`
`GetViewportExt()`	`GetViewportExtEx()`
`GetViewportOrg()`	`GetViewportOrgEx()`
`GetWindowExt()`	`GetWindowExtEx()`
`GetWindowOrg()`	`GetWindowOrgEx()`
`OffsetViewportOrg()`	`OffsetViewportOrgEx()`
`OffsetWindowOrg()`	`OffsetWindowOrgEx()`
`ScaleViewportExt()`	`ScaleViewportExtEx()`
`ScaleWindowExt()`	`ScaleWindowExtEx()`
`SetViewportExt()`	`SetViewportExtEx()`
`SetViewportOrg()`	`SetViewportOrgEx()`
`SetWindowExt()`	`SetWindowExtEx()`
`SetWindowOrg()`	`SetWindowOrgEx()`
`GetMetafileBits()`	`GetMetafileBitsEx()`
`SetMetafileBits()`	`SetMetafileBitsEx()`
`GetCurrentPosition()`	`GetCurrentPositionEx()`
`MoveTo()`	`MoveToEx()`
`DeviceCapabilities()`	`DeviceCapabilitiesEx()`
`DeviceMode()`	`DeviceModeEx()`

continues

TABLE 15.5 Continued

Windows 3.x Function	*Win32 Replacement*
`ExtDeviceMode()`	`ExtDeviceModeEx()`
`FreeSelector()`	N/A
`AllocSelector()`	N/A
`ChangeSelector()`	N/A
`GetCodeInfo()`	N/A
`GetCurrentPDB()`	`GetCommandLine()` and/or `GetEnvironmentStrings()`
`GlobalDOSAlloc()`	N/A
`GlobalDOSFree()`	N/A
`SwitchStackBack()`	N/A
`SwitchStackTo()`	N/A
`GetEnvironment()`	(Win32 file I/O functions)
`SetEnvironment()`	(Win32 file I/O functions)
`ValidateCodeSegments()`	N/A
`ValidateFreeSpaces()`	N/A
`GetInstanceData()`	N/A
`GetKBCodePage()`	N/A
`GetModuleUsage()`	N/A
`Yield()`	`WaitMessage()` and/or `Sleep()`
`AccessResource()`	N/A
`AllocResource()`	N/A
`SetResourceHandler()`	N/A
`AllocDSToCSAlias()`	N/A
`GetCodeHandle()`	N/A
`LockData()`	N/A
`UnlockData()`	N/A
`GlobalNotify()`	N/A
`GlobalPageLock()`	`VirtualLock()`

TABLE 15.6 Win32 API Compatibility Function

Windows 3.x Function	*Win32 Replacement*
`DefineHandleTable()`	N/A
`MakeProcInstance()`	N/A

Windows 3.x Function	*Win32 Replacement*
`FreeProcInstance()`	N/A
`GetFreeSpace()`	`GlobalMemoryStatus()`
`GlobalCompact()`	N/A
`GlobalFix()`	N/A
`GlobalUnfix()`	N/A
`GlobalWire()`	N/A
`GlobalUnwire()`	N/A
`LocalCompact()`	N/A
`LocalShrink()`	N/A
`LockSegment()`	N/A
`UnlockSegment()`	N/A
`SetSwapAreaSize()`	N/A

Concurrent 16-Bit and 32-Bit Projects

This section gives you some guidelines for developing projects that compile under 16-bit Delphi 1 or any 32-bit version of Delphi. Although you can follow the directions outlined in this chapter for source code compatibility, here are some further pointers to help you along:

- The `WINDOWS` conditional is defined by the compiler under Delphi 1; the `WIN32` conditional is defined under 32-bit versions of Delphi. You can use these defines to perform conditional compilation with the `{$IFDEF WINDOWS}` and `{$IFDEF WIN32}` directives.
- Avoid the use of any component or feature in 32-bit Delphi that isn't supported by Windows 3.1 or Delphi 1 if you want to recompile with Delphi 1 for a 16-bit application. For example, avoid the use of Win95 components and features such as multithreading that aren't available in Windows 3.1. The easiest way to ensure compatibility with projects intended for both Delphi 1 and 32-bit versions of Delphi is to develop the project in Delphi 1 and recompile it with 32-bit Delphi for optimized 32-bit performance.
- Be wary of differences between the APIs. If you have to use an API procedure or function that's implemented differently in the different platforms, make use of the `WINDOWS` and `WIN32` conditional defines.
- Each new version often adds more properties or different properties than those found in the previous version. This means, for example, that when version 5 components are saved to a DFM file, these new properties are written as well. Although it's often possible to "ignore" the errors that occur when loading projects with these properties in Delphi 1, it's often a more favorable solution to maintain two separate sets of DFM files, one for each platform.

Win32s

One other option for leveraging a single code base into both 16- and 32-bit Windows is to attempt to run your 32-bit Delphi applications under Win32s. Win32s is an add-on to Windows 3.*x*, which enables a subset of the Win32 API to function on 16-bit Windows. One serious drawback to this method is that many Win32 features, such as threads, are not available under Win32s (this precludes your use of the Borland Database Engine in this circumstance because the BDE makes use of threads). If you choose this route, you should also bear in mind that Win32s is not an officially supported platform for 32-bit Delphi, so you're on your own if things don't quite function as you expect.

Summary

Armed with the information provided by this chapter, you should be able to migrate your projects smoothly from any previous version of Delphi to Delphi 5. Also, with a bit of work, you'll be able to maintain projects that work with multiple versions of Delphi.