An Overview of the Delphi 2.0 Optimizing Native Code Compiler for Windows 95 and NT

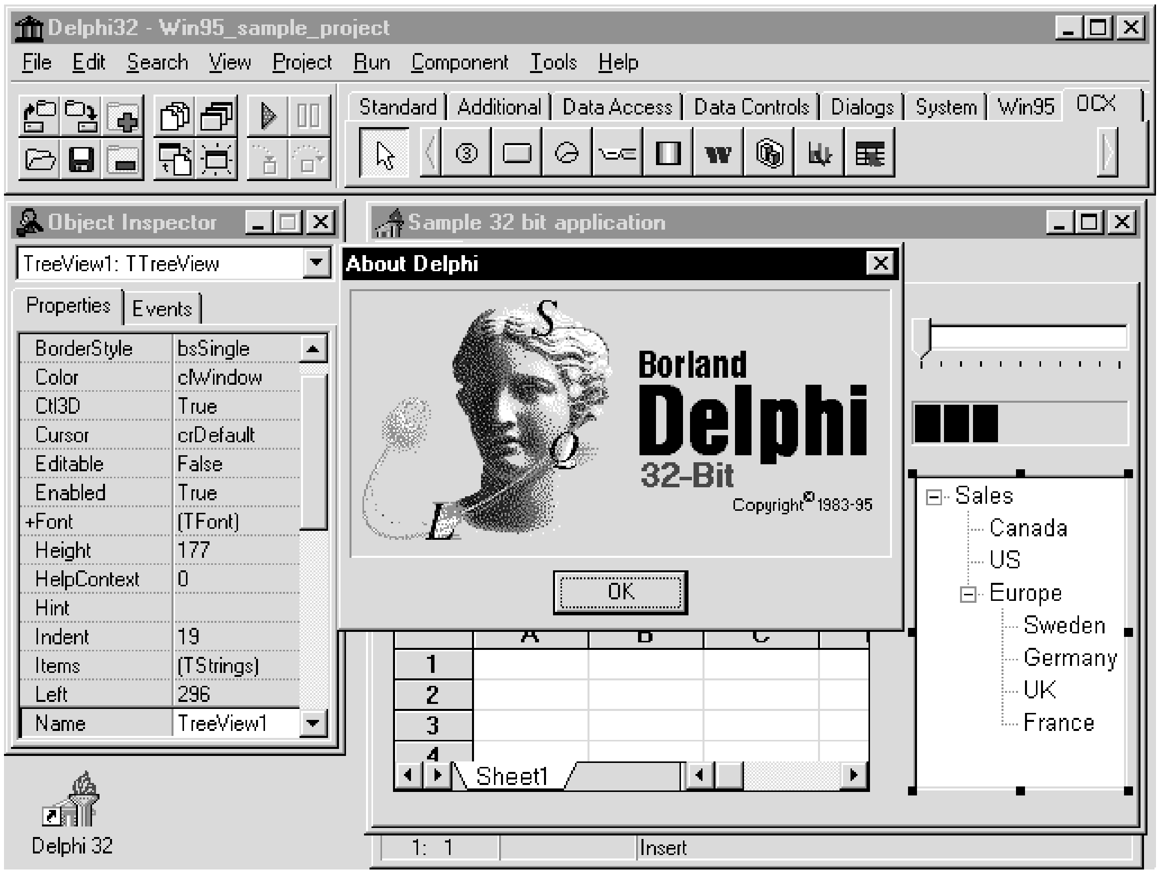
Zack Urlocker
Director of Delphi Product Management

Borland International
1/3/96

Overview

Since the introduction in February 1995, Borland's Delphi and Delphi Client/Server development tools have set a new standard in high-performance rapid application development. As a result of Delphi's unique combination of a native code compiler, visual two-way tools and scaleable database technology, Delphi has won dozens of awards worldwide and has become the fastest growing visual tool. Delphi has also achieved a tremendous level of third party support including dozens of add-on libraries and compatible tools, over 30 books, a half dozen monthly magazines and newsletters and many training courses supported by a growing number of third party consultants.

Borland has developed a new second generation 32-bit version of Delphi known as Delphi 2.0. This new 32-bit release incorporates several new technologies in order to further improve the productivity of developers and the performance of their applications. Delphi 2.0 is based on a new 32-bit optimizing native code compiler offering even greater performance and larger memory capacity than before. In addition, the new compiler architecture helps programmers write correct code by offering better error checking and smarter module management. Delphi 2.0 also takes advantage of the latest OLE technology including OLE controls (OCXs) and OLE automation.


Delphi32 - Win95_sample_project
File Edit Search View Project Run Component Tools Help
Standard Additional Data Access Data Controls Dialogs System Win95 OCX
Object Inspector
TreeView1: TTreeView
Properties Events
Sample 32 bit application
About Delphi
Borland
Delphi
32-Bit
Copyright© 1983-95
OK
Sales
Canada
US
Europe
Sweden
Germany
UK
France
A B C
1
2
3
4
Sheet1
1: 1
Insert
Delphi 32

BorderStyle	bsSingle
Color	clWindow
Ctl3D	True
Cursor	crDefault
Editable	False
Enabled	True
+Font	(TFont)
Height	177
HelpContext	0
Hint	
Indent	19
Items	(TStrings)
Left	296
Name	TreeView1

Figure 1 -- Delphi 2.0 supports all of the features of both Windows 95 and Windows NT.

This whitepaper focuses on explaining the features and benefits of the new 32-bit optimizing native code compiler. Delphi 2.0 will also include many other improvements to take advantage of platform features found on Windows 95 and Windows NT. Delphi 2.0 includes complete support for the Windows 95 user interface, including many new components, support for the entire Windows 95 API, including advanced features like multi-threading, Unicode and MAPI. Delphi 2.0 meets all of the requirements to obtain the Windows 95 logo, and makes it easy for developers to obtain the logo for their own applications.

Our goals for Delphi 2.0 are:

- Further increase the performance advantage with a new 32-bit optimizing compiler
- Increase scalability for Client/Server applications with the 32-bit Borland Database Engine
- Increase the ability to reuse objects by supporting OLE controls (OCX) and OLE automation
- Provide full support for all Windows 95 user interface elements and APIs
- Fully support 32-bit development and deployment on both Windows 95 and Windows NT
- Meet the Windows 95 logo requirements
- Provide full compatibility for existing Delphi applications

Exact packaging, pricing and availability of Delphi 2.0 will be provided at a later date.

Delphi Development Today

Delphi customers include such organizations as Alcatel, American Stores, Arthur Anderson, AT&T, BMW, BP Shipping, Bank of America, BBC Television, British Telecom, City of Los Angeles, Compaq, Conoco, Coopers & Lybrand, DHL, Dover Elevators, EDS, Ernst & Young, Fiat, First National Bank of Chicago, Glaxo, KPMG, Mercury Communications, Netscape, Sarah Lee Knitting, Standard & Poors, SwissBank SG Warburg, Union Bank, US Marine Corps and many others. Delphi and Delphi Client/Server are being used for a broad range of applications that demand rapid development and high performance. Sample applications include:

- A leading petrochemical company has created an executive information system to provide key operating and financial information on an hourly basis to over 600 users worldwide.
- A commercial bank has created a global funds transfer system supporting 25 currencies with secure transactions between major financial centers worldwide.
- A marketing research agency has created an executive information system to provide on-line analysis of sales data stored on an Oracle server.
- A rental company uses Delphi Client/Server to create an application that tracks equipment nationwide across several locations with more than forty simultaneous users connecting to an InterBase server running on Windows NT.
- A major textile manufacturer has used Delphi Client/Server to develop an all new IS infrastructure for all accounting, manufacturing and sales information that will access an Oracle database with over 100 million rows.
- An Internet consulting group used Delphi Client/Server to create an application being used by advertising agencies, retailers and banks to manage pages on the World Wide Web.
- A publisher of technical books and magazines has used Delphi to create a full motion multi-media application to break into software publishing.
- A national health agency use Delphi to create a Dynamic Link Library (DLL) to extend an existing Paradox for Windows application in order to perform real-time data collection through scientific instruments for measuring Radon gas levels.
- A Fortune 100 organization is using a Delphi Client/Server application and Microsoft SQL as part of a major business re-engineering effort to dramatically increase efficiency and increase customer satisfaction.
- A major industrial commodity markets analyst has created a global client/server system using Delphi Client/Server to access a 200 gigabyte Oracle database to produce up-to-date daily reports that will be deployed to thousands of customer worldwide.

Several case studies are available separately that describe these and other applications in more detail.

New 32-Bit Optimizing Native Code Compiler Architecture

Summary: Delphi 2.0 includes an all new optimizing native code compiler featuring:
- *a common backend with C++*
- *new compiler optimizations resulting in up to 300% - 400% performance improvements*
- *new faster, optimizing linker*
- *EXEs which are 20-25% smaller than before and still require no runtime interpreter DLLs*
    - *new support for OBJ files for easier code sharing with C*

The new 32-bit release of Delphi is built on an all new 32-bit optimizing native code compiler architecture. Borland is leveraging more than ten years of leading edge compiler technology in this new release. The new compiler is built using a common compiler “back end” technology that is shared with the award-winning Borland C++ compiler. This means that not only do Delphi developers get an even greater performance advantage over p-code interpreter systems, it is much easier to share code between Delphi and C++ since both can use the standard OBJ object file format. In addition, because Delphi 2.0 is a native code compiler, there are still no runtime interpreter DLLs required when deploying Delphi applications.

Preliminary Benchmark results

All benchmark tests were performed on a Gateway 2000 V66 (66Mhz 486 processor) with 16 megabytes of memory. The 16 bit benchmarks were performed using on Windows 3.1. The 32-bit benchmarks were performed with a pre-release version of Delphi 2.0.

Larger numbers indicate faster performance

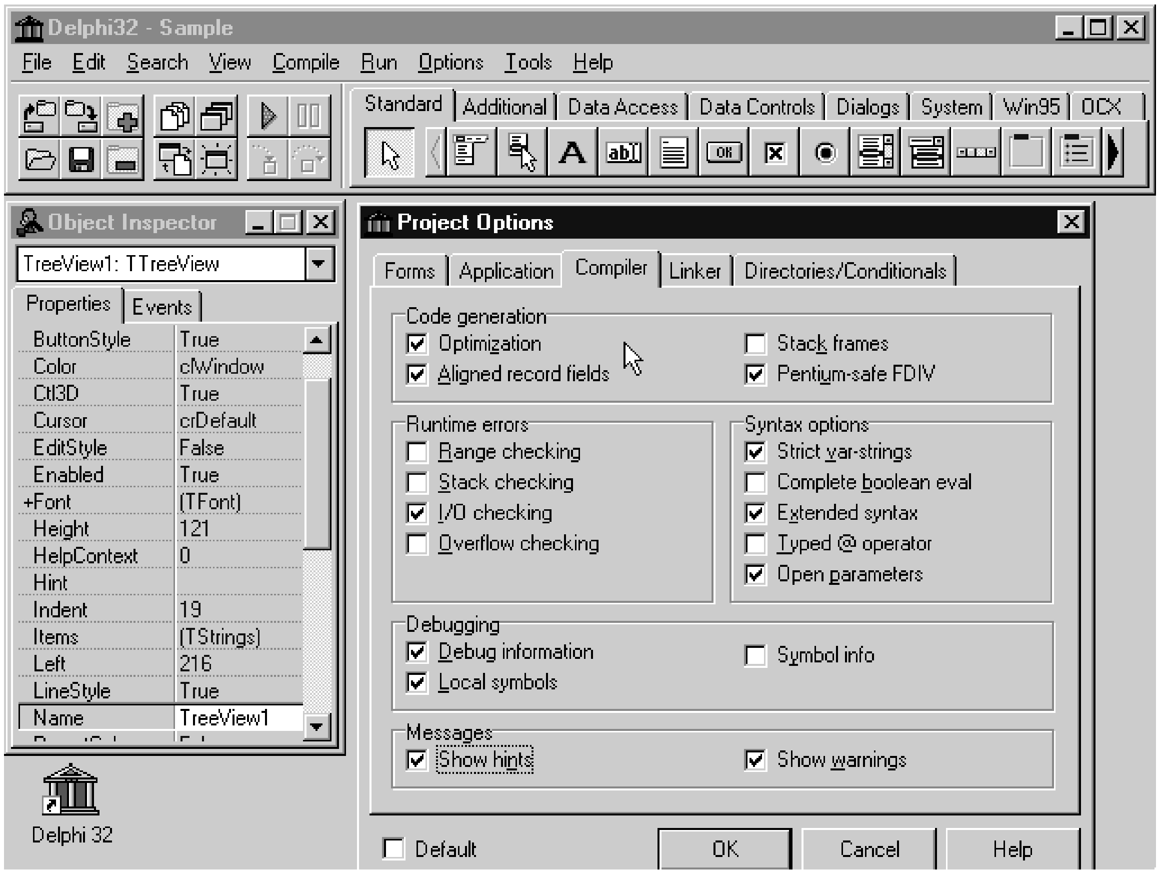
Sieve	0.22	11.95	52.77	179.37
Whetstone	0.04	1.41	4.70	15.53
File write	0.05	0.42	0.74	2.89
File read	0.05	0.33	1.75	5.28

Benchmark tests show that code compiled with Delphi 2.0 applications can run from approximately 300% to nearly 400% faster than 16 bit Delphi applications. This means that Delphi continues to expand it’s performance advantage of over 10-20 times faster than p-code interpreters. For example, Delphi 2.0 Sieve benchmark results are 15 times faster than VB 3.0 and 815 times faster than PowerBuilder 3.0. In addition, because of the new optimizing linker, .EXEs are 20-25% smaller than before and still require no runtime interpreter DLLs.

32-Bit Compiler Optimizations

The new 32-bit native code compiler achieves it’s performance increase by using a number of new code optimization techniques. In the past, optimizing compilers often required experimentation with complex compiler directives to achieve the fastest performance. In addition, they often depended on performing additional “passes” over the code which slows down the compiler and discourage rapid application development. Delphi 2.0 uses many new optimization techniques automatically, without guesswork. In addition, the compiler remains the fastest native code compiler in the world performing at over 350,000 lines per minute on a pentium. As a result, programmers always get the benefit of rapid application development and high performance.
The new 32-bit native code compiler includes a number of automatic optimizations including register optimizations, call stack overhead elimination, common subexpression elimination, loop induction

variables which result in faster performance for much code. All of the optimizations performed in the 32-bit compiler are guaranteed correct and in no way change the meaning of the code. The optimizations can also be disabled for benchmark comparisons. Delphi 2.0 also includes the capability of generating "Pentium-safe FDIV" code to guarantee that Delphi applications which use floating point division run correctly even on the so-called "flawed Pentium" processors.


Delphi32 - Sample
File Edit Search View Compile Run Options Tools Help
Standard Additional Data Access Data Controls Dialogs System Win95 OCX
Object Inspector
TreeView1: TTreeView
Properties Events
Delphi 32
Project Options
Forms Application Compiler Linker Directories/Conditionals
OK Cancel Help

ButtonStyle	True
Color	clWindow
Ctl3D	True
Cursor	crDefault
EditStyle	False
Enabled	True
+Font	(TFont)
Height	121
HelpContext	0
Hint	
Indent	19
Items	(TStrings)
Left	216
LineStyle	True
Name	TreeView1

Code generation
- [✓] Optimization
- [✓] Aligned record fields
- [ ] Stack frames
- [✓] Pentium-safe FDIV

Runtime errors
- [ ] Range checking
- [ ] Stack checking
- [✓] I/O checking
- [ ] Overflow checking

Syntax options
- [✓] Strict var-strings
- [ ] Complete boolean eval
- [✓] Extended syntax
- [ ] Typed @ operator
- [✓] Open parameters

Debugging
- [✓] Debug information
- [✓] Local symbols
- [ ] Symbol info

Messages
- [✓] Show hints
- [✓] Show warnings

- [ ] Default

Figure 2 -- Delphi 2.0's optimizing compiler creates native code that is 15-50x faster than p-code

Register Optimizations

Heavily used variables and parameters are automatically be placed into CPU registers thereby reducing the number of machine instructions required to access a variable. This results in faster and more compact code since there is no need to load variables from memory into registers. This optimization is done automatically by the compiler with no need to specify that certain variables or parameters should be placed in registers. The compiler also automatically performs variable "lifetime analysis" in order to reuse registers. For example, if a variable I is used exclusively in one section of code and a variable J is used exclusively in later section of code, the compiler uses a single register for both I and J.

Call Stack Overhead Elimination

When possible, parameters passed to functions or procedures are placed in CPU registers also. Not only does this eliminate the memory access similar to the earlier described register optimization, but it also means that there is no need to setup a "stack frame" in which to store the values temporarily. This eliminates instructions to create and destroy the stack frame so that function calls are highly efficient.

Common Subexpression Elimination

As the compiler translates complex mathematical expressions, it ensures that any common subexpressions, that is computations that would be performed more than once, are eliminated. This allows the programmer to write code in a manner that is clear and easy to read, knowing that the compiler automatically reduces it to its most compact and efficient form.

Loop Induction Variables

The compiler automatically uses loop induction variables as a way to speed up access to arrays or strings within loops. If a variable is used only to index into an array, for example in a **for** loop, the compiler "induces" the variable, eliminating multiplication operations and replacing them with a pointer that is incremented to access items in the array. In addition, if the element size is a 1, 2, 4 or 8 bytes, Intel scale indexing is used to provide additional performance benefits.

Example of Optimizations

The code below illustrates the compiler's use of register variables and loop induction. The expression Sums[X-1] is reduced to an induction variable - a register that is incremented in parallel with the loop variable X, instead of being recalculated on each iteration.

```
program example1;
uses Windows;

var
  Sums : array [0..500] of Integer;
  X: Integer;
begin
  Sums[0] := 0;
  for X := 1 to High(Sums) do
    Sums[X] := Sums[X-1] + X;
  writeln(Sums[High(Sums)]);
end.
```

The code that the new 32-bit compiler generates for this program as shown with Turbo Debugger:

```
ex1.9:  Sums[0] := 0;
    xor    eax,eax
    mov    [ex1.Sums],eax
ex1.10:  for X := 1 to High(Sums) do
    mov    edx,00000001      ; EDX = X, the loop control variable
    mov    eax,0040242C      ; EAX = the base memory address of Sums
ex1.11:  Sums[X] := Sums[X-1] + X;
    mov    ecx,[eax]         ; Get the value stored at Sums[X-1]
    add    ecx,edx           ; Add X to that value
    mov    [eax+04],ecx      ; Store the sum in Sums[X]
    inc    edx               ; increment the loop control variable
    add    eax,00000004      ; Increment the induction variable
    cmp    edx,000001F5      ; Test for loop end
    jne    ex1.11 (0040185D)
ex1.12:  writeln(Sums[High(Sums)]);
    mov    edx,[EX1.00402BFC]
    mov    eax,00402238
    call   @Write0Long
    call   @WriteLn
    call   @_IOTest
ex1.13: end.
```

New Optimizing Linker

As part of the compilation process, Delphi uses a new 32-bit linking technology that also includes several optimizations to operate faster. The new linker is 20% to 50% faster than previously due to a new unit caching scheme. This means that after the first time you compile an application, any forms or units that have not changed are linked directly in memory rather than from disk. In addition, .EXEs are 20-25% smaller than before and still require no runtime interpreter DLLs.

The linker also uses smarter version checking on units to eliminate the need to recompile units whenever possible, resulting in faster turnaround time and greater version resilience among libraries. For example, suppose that a function changes in a library unit that is used by four different forms. In the past, all four forms would have to be recompiled because the compiled code stored on disk (e.g. the DCU file) was an exact binary snapshot of what the compiler produced and, as a result, a change to any symbol would mean that the unit and all users of the unit would need to be recompiled. Now with the smarter linking and built-in version checking, the compiled code in the DCU file has a more robust, resilient format so that only units that used the modified function would need to be recompiled. This also makes it much easier for third party vendors to distribute compiled code without the need to recompile their libraries when a new version of Delphi is released.

In addition, Delphi 2.0 supports the OBJ file format so that you can more easily share code between Delphi and C or C++, in addition to being able to create and share DLLs. Since many C function libraries are available in OBJ format, this increases the number of third party libraries that can be used with Delphi.

Increased 32-Bit Capacity

The new 32-bit native code compiler runs in a 32-bit flat address space so that it completely eliminates all limitations previously associated with the 16-bit segmented architecture of Windows 3.1. For programmers this means it is now possible to take full advantage of all physical memory of the machine without resorting to direct Windows API calls. For example, you can declare arrays, strings, records and other data structures to be as large as you like, limited only by the operating system's limits. For example, on Windows 95 you can create strings up to 2 gigabytes. This is a great increase over the segment limitations in the 16 bit version of Delphi where data structures were limited to 64K.

New 32-Bit Data Types

The new 32-bit native code compiler also introduces several new data types to take advantage of the larger 32-bit flat address space, with additional flexibility and easy migration of 16-bit code.

These new data types include:
- long strings -- limited only by operating system memory
- wide strings -- double byte Unicode strings for internationalization of applications
- wide characters -- double byte Unicode character types for internationalization
  - variants -- provides the ability to change the type of a variable at runtime for more flexible database and OLE automation (see below for an example)

Other compiler enhancements

The Delphi 2.0 compiler also includes these additional enhancements:
- An expanded Open Tools API for integrating closely with version control software, CASE tools etc.
- Ability to create and inherit from COM (Common Object Model) objects for use in C++ and Delphi
- New **stdcall** reserved words for easier exporting of functions in DLLs
- Support for a new currency type for BCD arithmetic for increased accuracy in financial applications
- Threadsafe libraries and thread-local storage for easier multi-threading programming
  - Single line comments using a double slash //, as in C++

Support for OLE Automation and OLE Controls (OCXs)

Summary: Delphi 2.0 includes complete support for OLE facilities on Windows 95 and NT such as:
- *the ability to create OLE automation controller and server applications*
- *full compatibility with forthcoming Network OLE and Remote Automation for partitioning*
- *the ability to use existing third party OLE controls (OCXs)*
  - *the ability to customize OCXs via inheritance*

The Importance of being OLE

Microsoft's OLE technology includes a variety of important capabilities for increasing the ability of developers to create more modular and integrated applications. The goal with Delphi 2.0 has been complete adherence to Microsoft system standards to ensure that developers can use Delphi 2.0 to create a diverse range of applications without limitations. Delphi 2.0 goes beyond simply adhering to Microsoft standards; our goal has been to make the use of OLE technology even easier by making it completely object oriented. As a result, OLE technology is completely integrated into Delphi 2.0 ensuring that compatibility with future technologies such as Network OLE are ensured.

The OLE support in Delphi 2.0 includes the ability to easily install and use OLE controls (OCXs) as well as the ability to easily create OLE automation controllers and servers. For maximum flexibility, Delphi 2.0 can create both in-process and out-of-process servers. By supporting both OLE automation controllers and servers, Delphi 2.0 is completely compatible with the forthcoming Network OLE technology as well as VB 4.0's remote automation technology, with the added advantage of faster performance. In addition, because Delphi 2.0 is a native code compiler, advanced developers are able to write their own OLE controls (OCXs) within Delphi 2.0 though this is more difficult than creating Delphi components.

Using OLE Automation

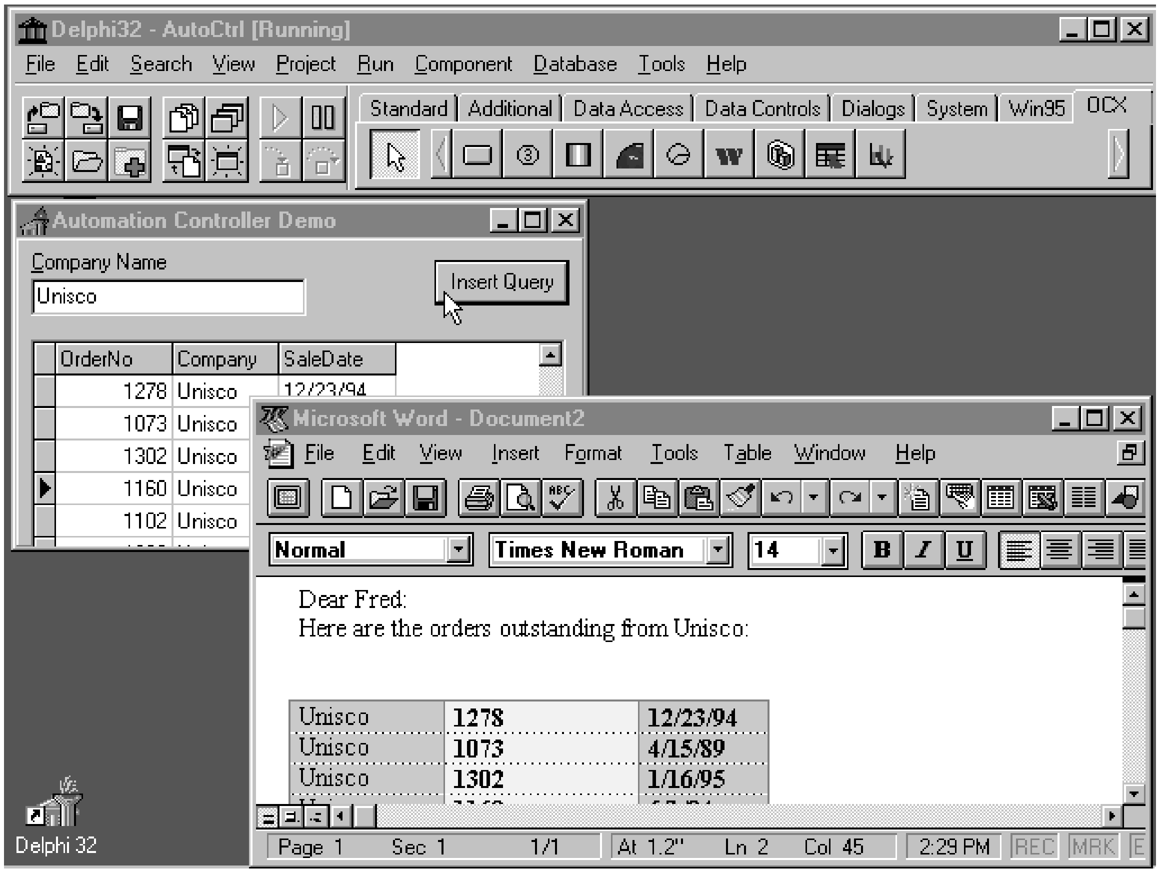
Delphi 2.0 makes use of a new type, called **variant**, to provide seamless integration of OLE automation. Delphi 2.0 allows you to create applications which can be either an OLE automation controller or an OLE automation server with equal ease. An OLE automation controller is the most common use of OLE automation among application developers and system integrators. For example, a Delphi 2.0 application can be used to control another OLE application, such as Word, Excel, Paradox, Quattro Pro and others.

The variant type allows developers to declare variables whose type is determined at runtime, allowing them to take advantage of the inherent flexibility of OLE automation. In effect, you can use a single variable to connect to different types of OLE automation servers at runtime. Delphi 2.0 also introduces the ability to use named parameters when making calls to OLE automation servers. For complex functions which often have dozens of parameters, developers can simply supply the parameters of interest and use the server's default values for the rest.

The code below is from a sample Delphi 2.0 application which performs a query and then inserts the result set into a Word document. Note that the OLE automation takes only four lines of code in addition to the declaration of the variable MSWord of type variant.

```
{ This example uses OLE automation to insert a query result into Word}
procedure TForm1.InsertBtnClick(Sender: TObject);
var
  MSWord: Variant;
  S: string;
  L: Integer;
begin
  { Connect to the automation server in MS Word and run the query}
  MSWord := CreateOleObject('Word.Basic');
  with Query1 do
  begin
```

```
    Close;
    Params[0].Text := Edit1.Text;
    Open;
    try
      First;
      L := 0;
      while not EOF do
      { Store the query result set in string S }
      begin
        S := S + Query1Company.AsString + ',' +
        Query1OrderNo.AsString + ',' + Query1SaleDate.AsString + #13;
        Inc(L);
        Next;
      end;
      { Use OLE automation to insert S into the Word document }
      MSWord.Insert(S);
      MSWord.LineUp(L, 1);
      MSWord.TextToTable(ConvertFrom := 2, NumColumns := 3);
    finally
      Close;
    end;
  end;
end;
```


Delphi32 - AutoCtrl [Running]
File Edit Search View Project Run Component Database Tools Help
Standard Additional Data Access Data Controls Dialogs System Win95 OCX
Automation Controller Demo
Insert Query
Microsoft Word - Document2
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
14
Page 1 Sec 1 1/1 At 1.2" Ln 2 Col 45 2:29 PM REC MRK
Delphi 32

Company Name: Unisco

OrderNo	Company	SaleDate
1278	Unisco	12/23/94
1073	Unisco	
1302	Unisco	
1160	Unisco	
1102	Unisco	

Dear Fred:
Here are the orders outstanding from Unisco:

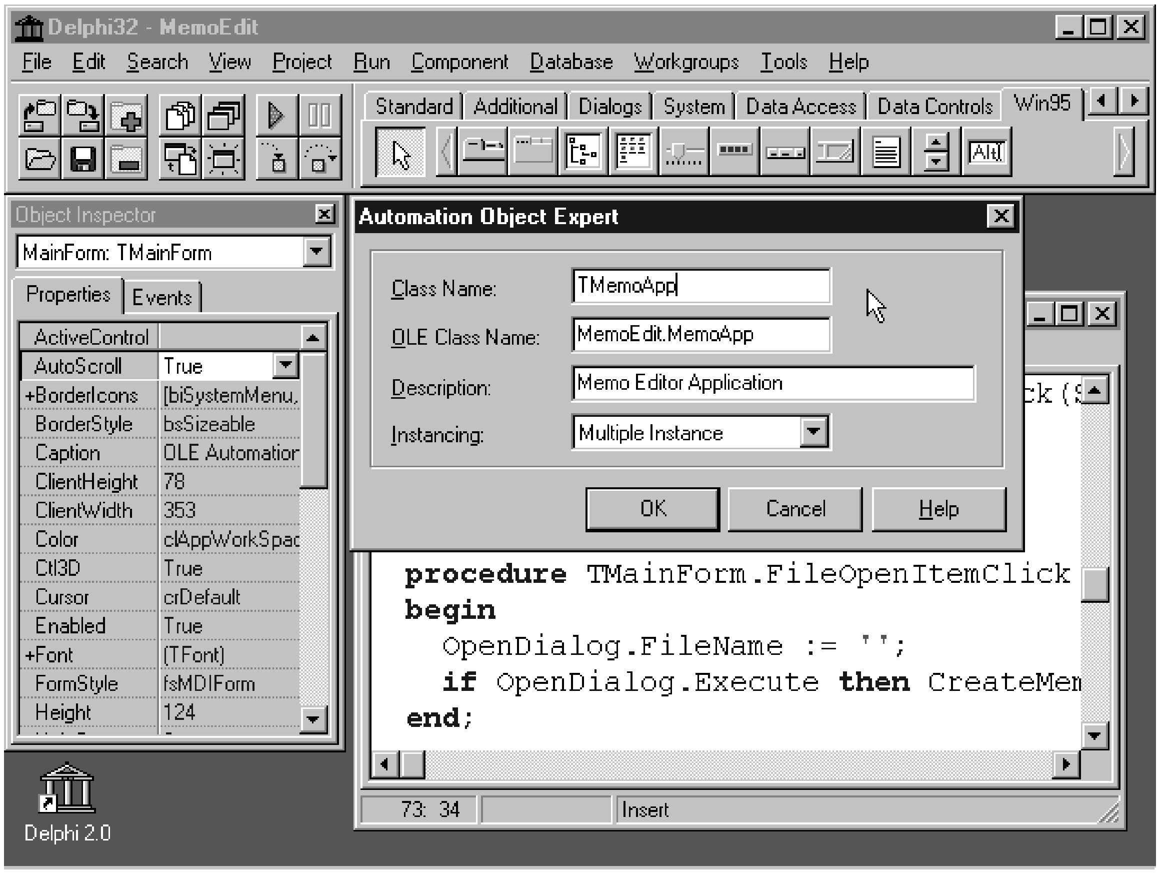
Unisco	1278	12/23/94
Unisco	1073	4/15/89
Unisco	1302	1/16/95

Figure 3 -- Use Delphi 2.0 to create OLE automation controllers and servers.

Creating OLE Automation Servers

Delphi 2.0 also enables you to create your own OLE automation servers. These can be either in-process or out-of-process (or local) servers. You can expose functions or methods of your application so that they can be called from other applications such as Microsoft Word, Excel, Visual Basic, C++, Paradox and Delphi 2.0. Because Delphi 2.0 can produce both OLE automation controllers and servers that are highly optimized native code executables, it offers a unique performance advantage that will be increasingly important with the emergence of Network OLE. Delphi 2.0 developers will be able to take advantage of having applications that are partitioned and have the fastest compiled code running on both the client and the server side.

To create a new OLE automation server you can use the Automation Object Expert. The expert automatically defines a new object that inherits from the TAutoObject type and sets up all of the OLE registration for you including the program ID, class ID and instancing options.


Delphi32 - MemoEdit
File Edit Search View Project Run Component Database Workgroups Tools Help
Standard Additional Dialogs System Data Access Data Controls Win95
Object Inspector
Properties Events
Automation Object Expert
OK Cancel Help
Delphi 2.0

MainForm: TMainForm

ActiveControl	
AutoScroll	True
+BorderIcons	[biSystemMenu,
BorderStyle	bsSizeable
Caption	OLE Automation
ClientHeight	78
ClientWidth	353
Color	clAppWorkSpac
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsMDIForm
Height	124

Class Name:	TMemoApp
OLE Class Name:	MemoEdit.MemoApp
Description:	Memo Editor Application
Instancing:	Multiple Instance

```
procedure TMainForm.FileOpenItemClick
begin
  OpenDialog.FileName := '';
  if OpenDialog.Execute then CreateMem
end;
```

73: 34 Insert

Figure 4 -- Delphi 2.0 lets you easily create OLE Automation servers that are Network OLE compatible

Then you can define the properties and methods you want to automate by adding them to the **automated** section of the object. The visibility of an identifier declared in an **automated** section is the same as a **public** identifier. You can expose properties, parameters and function results of any of the following types: Smallint, Integer, Single, Double, WordBool, Boolean, Currency, TDateTime, String and Variant.

The example code below shows how you can use the **automated** section to allow OLE controller applications to control a memo editor. You could then create either an in-process automation server (e.g. a DLL) or out-of-process OLE automation server (e.g. an .EXE).

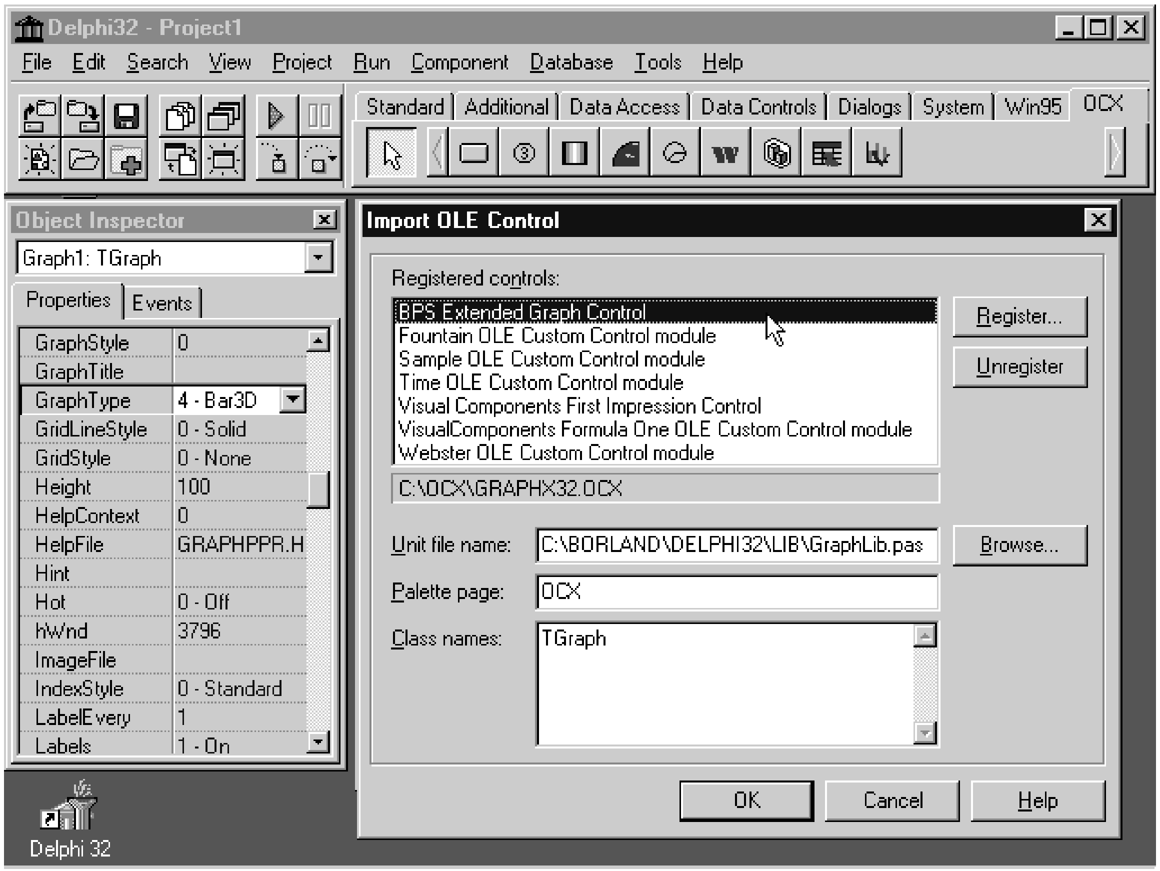
```
{ Shows the use of OLE automation server capabilities in Delphi 2.0 }
```

```
unit MemoAuto;

interface

uses
  OleAuto;

type
  { TMemoApp defines the automation server object and its services}
  TMemoApp = class(TAutoObject)
  private
    function GetMemo(Index: Integer): Variant;
    function GetMemoCount: Integer;
  automated
  { OLE enable the following properties and functions }
    procedure CascadeWindows;
    function NewMemo: Variant;
    function OpenMemo(const FileName: string): Variant;
    procedure TileWindows;
    property MemoCount: Integer read GetMemoCount;
    property Memos[Index: Integer]: Variant read GetMemo;
  end;

implementation

...

{ The registration info is created by the Automation Object Expert.}
procedure RegisterMemoApp;
const
  AutoClassInfo: TAutoClassInfo = (
    AutoClass: TMemoApp;
    ProgID: 'MemoEdit.Application';
    ClassID: '{F7FF4880-200D-11CF-BD2F-0020AF0E5B81}';
    Description: 'Memo Editor Application';
    Instancing: acSingleInstance);
begin
  Automation.RegisterClass(AutoClassInfo);
end;

initialization
  RegisterMemoApp;
end.
```

Using OLE Controls (OCXs)

Because Delphi 2.0 is a completely object-oriented environment, integration of OLE controls is seamless. You can install third party OLE controls (OCXs) just as you can install components you write in the Delphi environment. Delphi 2.0 gives you complete access to the OLE system registry so that you can load OLE controls and register them in one simple dialog.

When you install an OLE control (OCX), Delphi automatically creates an object wrapper to provide a completely object-oriented view of the control. Delphi 2.0 is the only rapid application development environment that makes OLE controls completely object-oriented. This makes it possible for developers to easily subclass any components, so that they can be easily customized through inheritance.


Delphi32 - Project1
File Edit Search View Project Run Component Database Tools Help
Standard Additional Data Access Data Controls Dialogs System Win95 OCX
Object Inspector
Properties Events
Import OLE Control
Register...
Unregister
Browse...
OK Cancel Help
Delphi 32

Graph1: TGraph

GraphStyle	0
GraphTitle	
GraphType	4 - Bar3D
GridLineStyle	0 - Solid
GridStyle	0 - None
Height	100
HelpContext	0
HelpFile	GRAPHPPR.H
Hint	
Hot	0 - Off
hWnd	3796
ImageFile	
IndexStyle	0 - Standard
LabelEvery	1
Labels	1 - On

Registered controls:

- BPS Extended Graph Control
- Fountain OLE Custom Control module
- Sample OLE Custom Control module
- Time OLE Custom Control module
- Visual Components First Impression Control
- VisualComponents Formula One OLE Custom Control module
- Webster OLE Custom Control module

C:\OCX\GRAPHX32.OCX

Unit file name: C:\BORLAND\DELPHI32\LIB\GraphLib.pas

Palette page: OCX

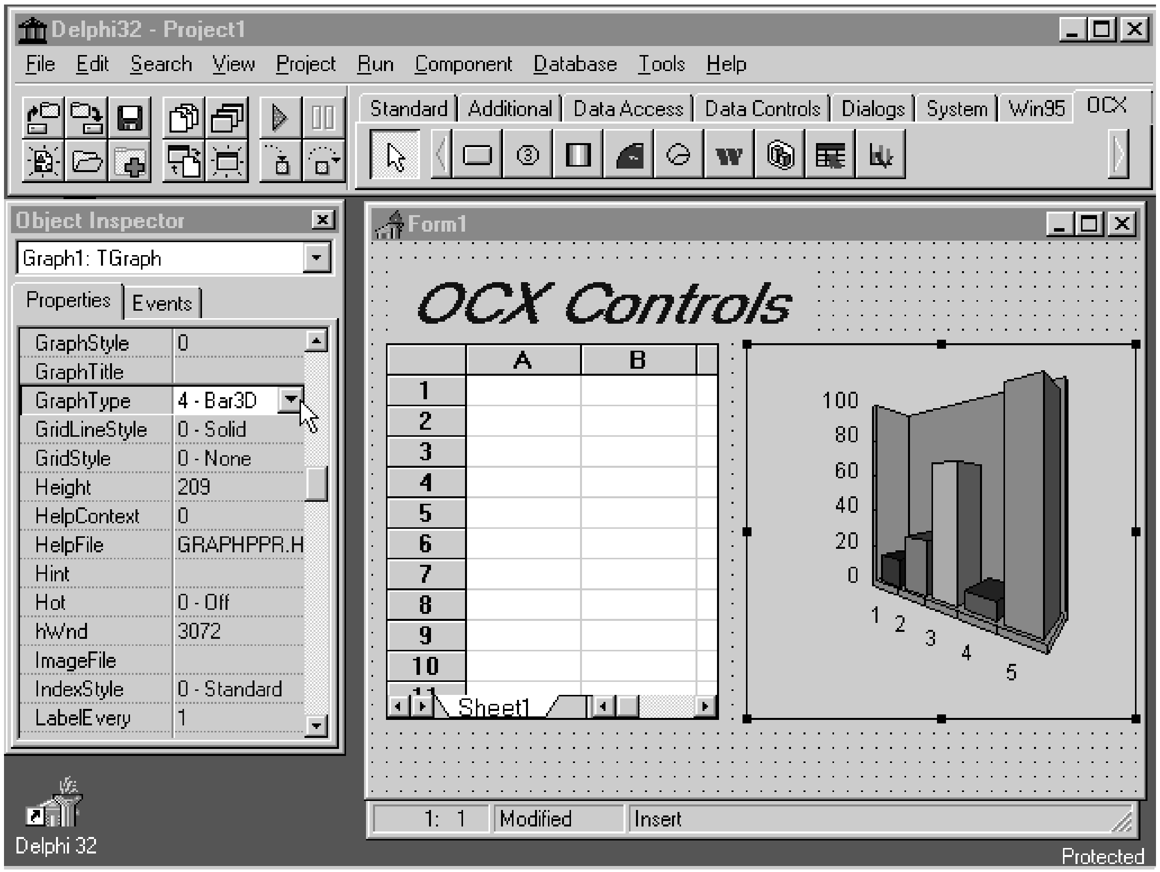
Class names: TGraph

Figure 5 -- Delphi 2.0 lets you easily install OLE controls as well as your own Delphi components

Below is an excerpt from the code that is automatically generated and compiled behind the scenes when you install an OLE control.

```
{ An object wrapper is automatically generated by Delphi when you install any OCX }
TGraph = class(TOleControl)
 private
   FOnHotHit: TGraphHotHit;
   function Get_Color(Index: Smallint): Smallint; stdcall;
   procedure Set_Color(Index: Smallint; Value: Smallint); stdcall;
   function Get_Data(Index: Smallint): Single; stdcall;
   procedure Set_Data(Index: Smallint; Value: Single); stdcall;
...
public
   property Color[Index: Smallint]: Smallint read Get_Color write Set_Color;
   property Data[Index: Smallint]: Single read Get_Data write Set_Data;
...
 published
   property TabOrder;
   property OnClick;
   property OnDblClick;
...
 end;
```

One an OLE control is installed into the Delphi 2.0 component palette, it can be used just like any of the supplied Delphi components. All of the OLE controls properties and events are fully accessible within the environment through the Object Inspector as shown below.



Graph1: TGraph

GraphStyle	0
GraphTitle	
GraphType	4 - Bar3D
GridLineStyle	0 - Solid
GridStyle	0 - None
Height	209
HelpContext	0
HelpFile	GRAPHPPR.H
Hint	
Hot	0 - Off
hWnd	3072
ImageFile	
IndexStyle	0 - Standard
LabelEvery	1

Figure 6 -- Delphi 2.0 allows you to customize third party OCXs via inheritance

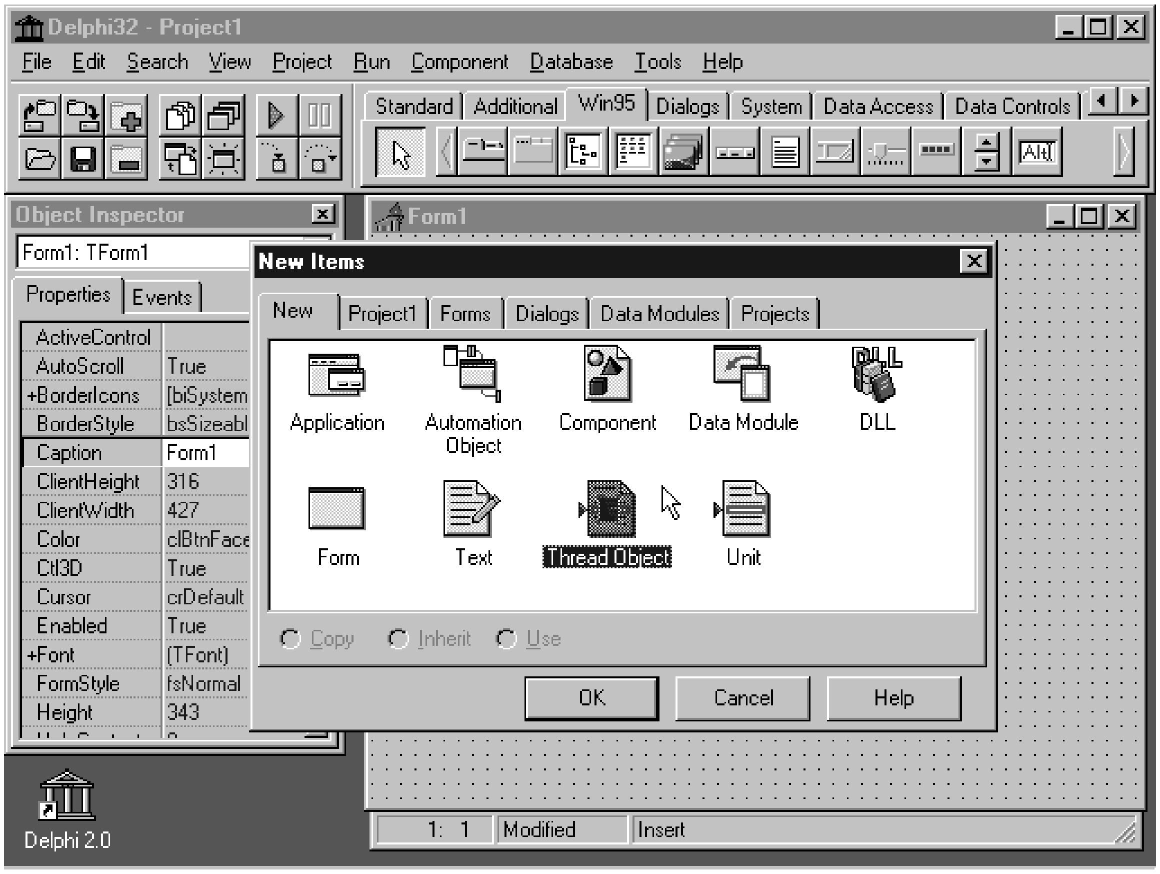
Taking Advantage of Windows 95 and NT Features

Because Delphi 2.0 is a native code compiler, it fully supports all of the platform features of Windows 95 and NT. It includes:

- *Multi-threading support*
- *Unicode double byte strings for localization support*
    - *MAPI*

Multi-threading

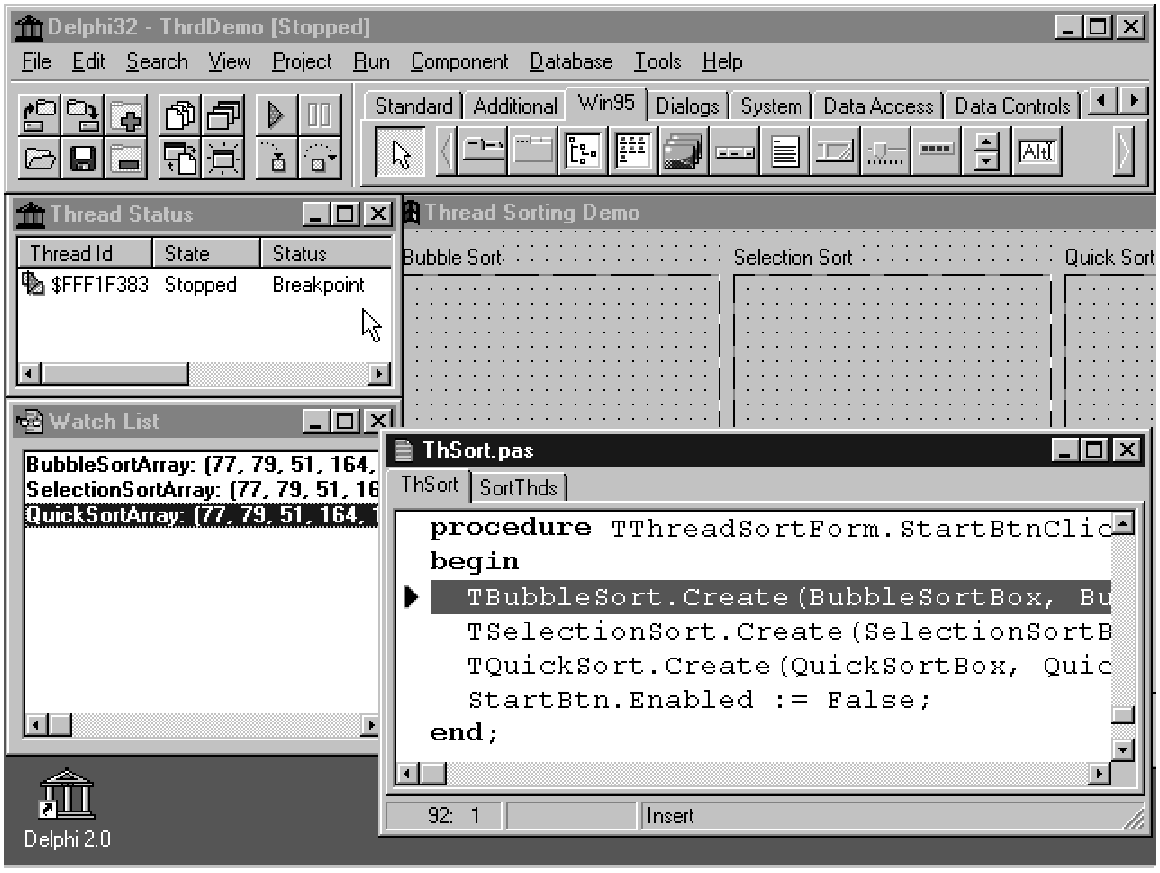
Because Delphi 2.0 is a native code compiler it can take advantage of any platform feature on Windows 95 or Windows NT. This includes complete support for features such as multi-threading. You can easily create multi-threaded applications by selecting New Thread Object from the Delphi Object Repository. This will generate a unit with an object that inherits from the TThread type to simplify the creation of multi-threaded applications. You also have direct access to the threading API. For example, you can call the CreateThread API function with an Object Pascal function as a parameter. Similarly, you can set a thread priority by calling the SetThreadPriority API function.



ActiveControl	
AutoScroll	True
+BorderIcons	[biSystem
BorderStyle	bsSizeabl
Caption	Form1
ClientHeight	316
ClientWidth	427
Color	clBtnFace
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsNormal
Height	343

Figure 7 -- Delphi 2.0 allows you to easily create multi-threading applications on Windows 95 and NT.

The Delphi Visual Component Library (VCL) includes support for creating threadsafe applications. For example, the Synchronize method of the TThread class allows you to ensure that manipulation of VCL components is done safely within a thread without any possibility of conflicts in any other thread. Delphi 2.0 also introduces a new reserved word **ThreadVar** which allows you to declare thread-local storage, giving you can complete control over variables used in different threads.


Delphi32 - ThrdDemo [Stopped]
File Edit Search View Project Run Component Database Tools Help
Standard Additional Win95 Dialogs System Data Access Data Controls
Thread Status
Thread Sorting Demo
Bubble Sort
Selection Sort
Quick Sort
Watch List
ThSort.pas
ThSort SortThds
Delphi 2.0
92: 1
Insert

Thread Id	State	Status
$FFF1F383	Stopped	Breakpoint

```
BubbleSortArray: (77, 79, 51, 164,
SelectionSortArray: (77, 79, 51, 16
QuickSortArray: (77, 79, 51, 164,
```

```
procedure TThreadSortForm.StartBtnClic
begin
  TBubbleSort.Create(BubbleSortBox, Bu
  TSelectionSort.Create(SelectionSortB
  TQuickSort.Create(QuickSortBox, Quic
  StartBtn.Enabled := False;
end;
```

Figure 8 -- Delphi 2.0's debugger includes support for viewing the status of all threads

The code shown below illustrates how you can use the TThread class to create a multi-threaded sorting program. There are three descendants of the TSortThread class, each of which defines its sort method.

```
{Example of multi-threading in Delphi 2.0 }
TSortThread = class(TThread)
  private
    FBox: TPaintBox;
    FSortArray: PSortArray;
    FSize: Integer;
    FA, FB, FI, FJ: Integer;
    procedure DoVisualSwap;
  protected
    procedure Execute; override;
    procedure VisualSwap(A, B, I, J: Integer);
    procedure Sort(var A: array of Integer); virtual; abstract;
  public
    constructor Create(Box: TPaintBox; var SortArray: array of
Integer);
  end;

  TBubbleSort = class(TSortThread)
  protected
    procedure Sort(var A: array of Integer); override;
  end;
```

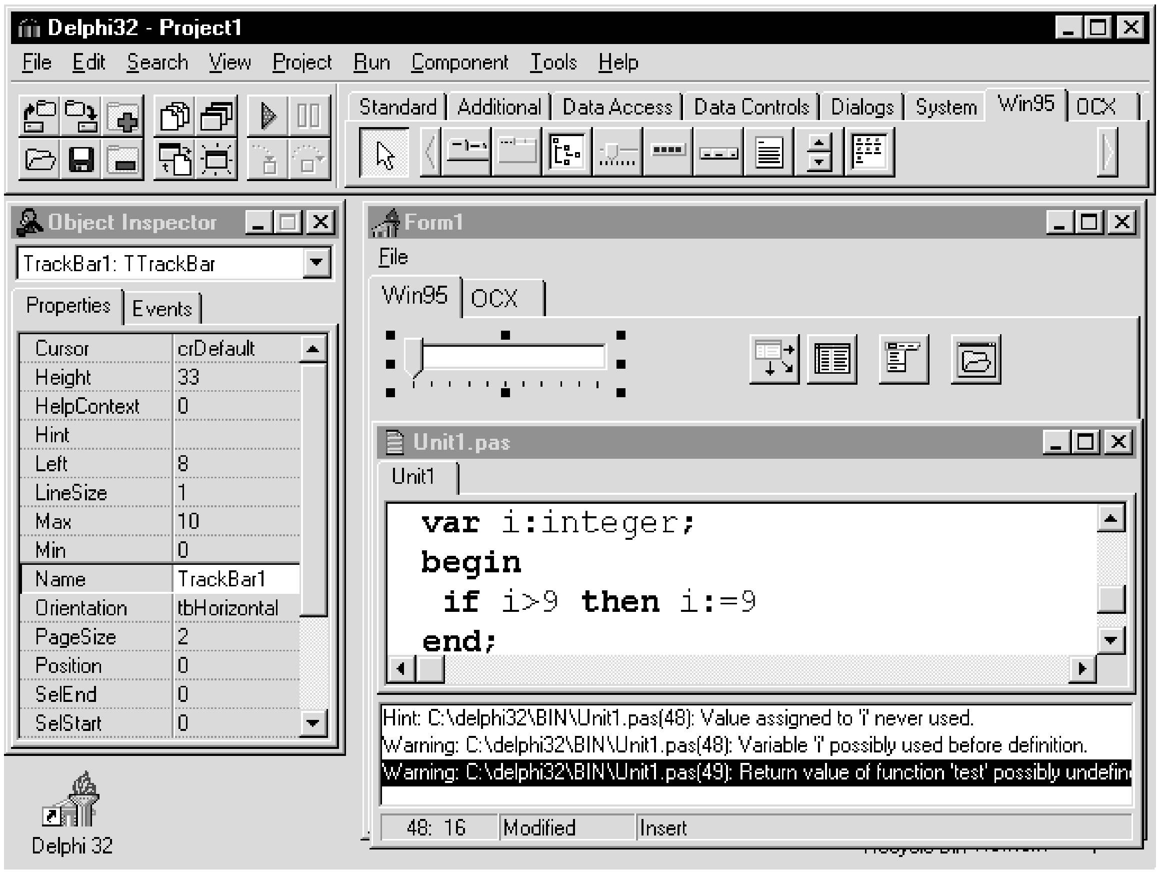
```
...

implementation

{ TSortThread }
constructor TSortThread.Create(Box: TPaintBox; var SortArray: array of
Integer);
begin
  inherited Create(False);
  FBox := Box;
  FSortArray := @SortArray;
  FSize := High(SortArray) - Low(SortArray) + 1;
end;

{ Since DoVisualSwap uses a VCL component (i.e., the TPaintBox) it
should never be called directly by this thread.  DoVisualSwap should be
called by passing it to the Synchronize method which causes
DoVisualSwap to be executed by the main VCL thread, avoiding multi-
thread conflicts.}
procedure TSortThread.DoVisualSwap;
begin
  with FBox do
  begin
    Canvas.Pen.Color := clBtnFace;
    PaintLine(Canvas, FI, FA);
    PaintLine(Canvas, FJ, FB);
    Canvas.Pen.Color := clRed;
    PaintLine(Canvas, FI, FB);
    PaintLine(Canvas, FJ, FA);
  end;
end;

{ VisusalSwap is a wrapper on DoVisualSwap making it easier to use.
The
  parameters are copied to instance variables so they are accessable
  by the main VCL thread when it executes DoVisualSwap }
procedure TSortThread.VisualSwap(A, B, I, J: Integer);
begin
  FA := A;
  FB := B;
  FI := I;
  FJ := J;
  Synchronize(DoVisualSwap);
end;

{ The Execute method is called when the thread starts }
procedure TSortThread.Execute;
begin
  Sort(Slice(FSortArray^, FSize));
end;

...

end;
```

Helping Programmers Write Correct Code

Summary: Delphi 2.0's new compiler architecture makes it easier to write correct code with:
- *Multi-error architecture*
- *New hints and warnings that detect common coding errors*
- *Improved error messages*
- *Smart module management*
    - *Automatic form linking*

Improved Compiler Error Messages and Diagnostics

One of the frequently overlooked advantages of a native code compiler is that it provides the programmer with a complete check of the program before running. Compilers can often catch logic errors that result from ambiguous or incorrect code that is not detected by an interpreter. Because the Object Pascal language is a strongly typed language, it prevents the programmer from many common errors of using types in an incorrect fashion. The new 32-bit compiler also has a "multi-error" architecture so that it can continue to compile code to find all of the errors, rather than stopping at the first error. This makes it easier to verify large programs for correctness.


Delphi32 - Project1
File Edit Search View Project Run Component Tools Help
Standard Additional Data Access Data Controls Dialogs System Win95 OCX
Object Inspector
TrackBar1: TTrackBar
Properties Events
Form1
File
Win95 OCX
Unit1.pas
Unit1
Delphi 32

Cursor	crDefault
Height	33
HelpContext	0
Hint	
Left	8
LineSize	1
Max	10
Min	0
Name	TrackBar1
Orientation	tbHorizontal
PageSize	2
Position	0
SelEnd	0
SelStart	0

```
var i:integer;
begin
  if i>9 then i:=9
end;
```

```
Hint: C:\delphi32\BIN\Unit1.pas(48): Value assigned to 'i' never used.
Warning: C:\delphi32\BIN\Unit1.pas(48): Variable 'i' possibly used before definition.
Warning: C:\delphi32\BIN\Unit1.pas(49): Return value of function 'test' possibly undefin
```

```
48: 16    Modified    Insert
```

Figure 9 -- Delphi 2.0 provides improved error messages and diagnostics.

In Delphi 2.0, we have improved the error messages and diagnostics of the compiler in order to provide hints and warnings when code could be incorrect.

These helps to eliminate many common errors including:

- using uninitialized variables and pointers
- unused variables
- unused function return value
- empty loops
  - type mismatch

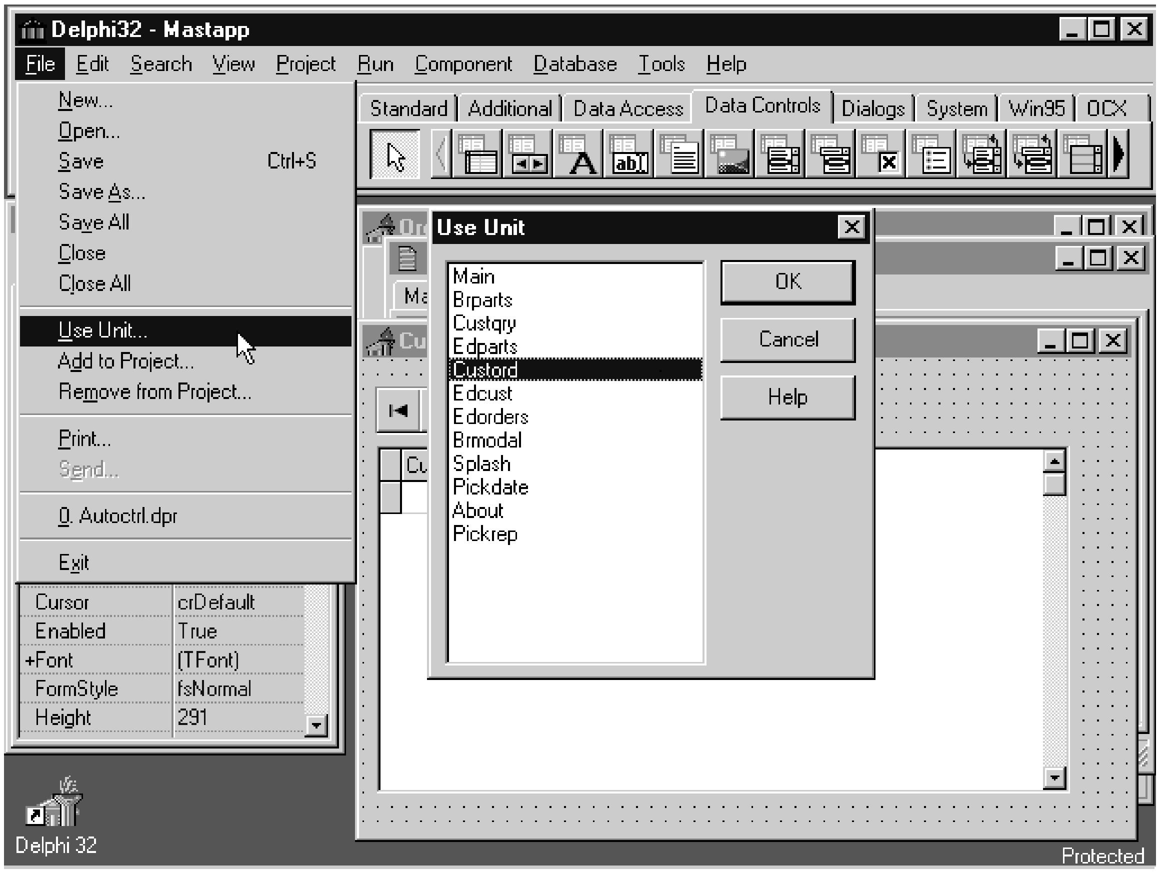
Improved error messages

The compiler also offers better diagnostic messages on syntactic errors, making Delphi easier for programmers who are new to Object Pascal. Rather than simply reporting a general "Error in statement" message, the new compiler gives a much clearer indication of the problem. This includes many common syntactic errors such as:

- forgotten semi-colons
  - semi-colons in front of the ELSE statement

Smart module management

Delphi has always used the concept of separate compilation of code modules, known as units, in order to facilitate development and testing of large projects. By having separate compilation of units you can enforce stricter discipline between units through interface sections rather than having to expose a large number of global variables which can lead to more difficult code maintenance. However, newcomers to Delphi often found that manually updating the **uses** statements was tedious. In Delphi 2.0, module management has been facilitated through the use of a new File Uses command and form linking.



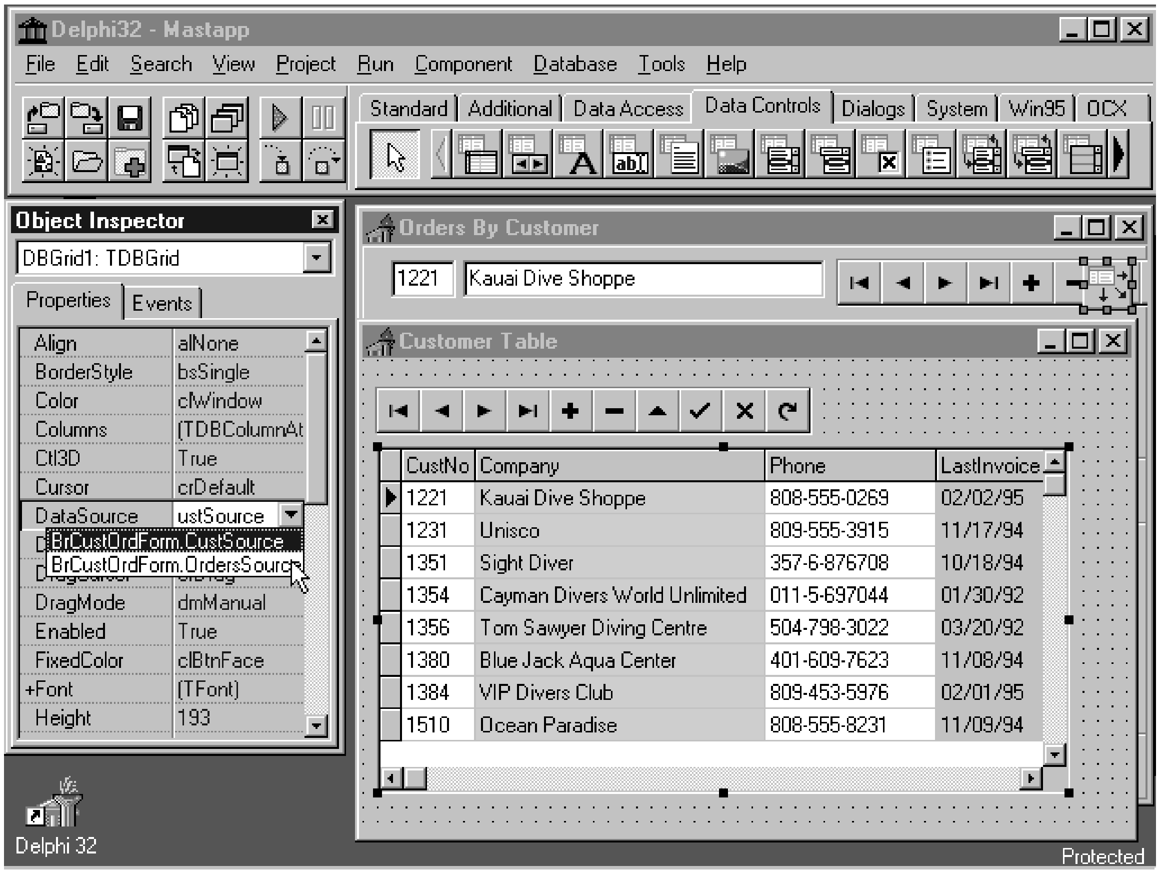
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsNormal
Height	291

Figure 10 -- Delphi 2.0 includes automatic module management for easier maintenance

In addition, the development environment is more intelligent now and recognizes when a **uses** statement should be updated automatically. For example, if you reference Form2 from Unit inside of Form1 you will be prompted to add Unit2 to your **uses** list when you recompile.

Automatic form linking

The next logical step to make modular programming easier was to allow automatic form linking across different modules. Although you have always been able to access the public objects, properties and code in different forms programmatically, you could not access them in the design environment. It is quite common to want to refer to data sources, queries and tables across different forms. Therefore in Delphi 2.0, it is now possible to link these components at design time across forms without having to write code. This makes it easier to create reusable modules that encapsulate data access and business rules separate from the user interface components.


Delphi32 - Mastapp
File Edit Search View Project Run Component Database Tools Help
Standard Additional Data Access Data Controls Dialogs System Win95 OCX
Object Inspector
DBGrid1: TDBGrid
Properties Events
Orders By Customer
Customer Table
Delphi 32
Protected

Align	alNone
BorderStyle	bsSingle
Color	clWindow
Columns	(TDBColumnAt
Ctl3D	True
Cursor	crDefault
DataSource	ustSource
	BrCustOrdForm.CustSource
	BrCustOrdForm.OrdersSourc
DragMode	dmManual
Enabled	True
FixedColor	clBtnFace
+Font	(TFont)
Height	193

1221 Kauai Dive Shoppe

CustNo	Company	Phone	LastInvoice
1221	Kauai Dive Shoppe	808-555-0269	02/02/95
1231	Unisco	809-555-3915	11/17/94
1351	Sight Diver	357-6-876708	10/18/94
1354	Cayman Divers World Unlimited	011-5-697044	01/30/92
1356	Tom Sawyer Diving Centre	504-798-3022	03/20/92
1380	Blue Jack Aqua Center	401-609-7623	11/08/94
1384	VIP Divers Club	809-453-5976	02/01/95
1510	Ocean Paradise	808-555-8231	11/09/94

Figure 11 -- Delphi 2.0 includes form linking to share components across forms

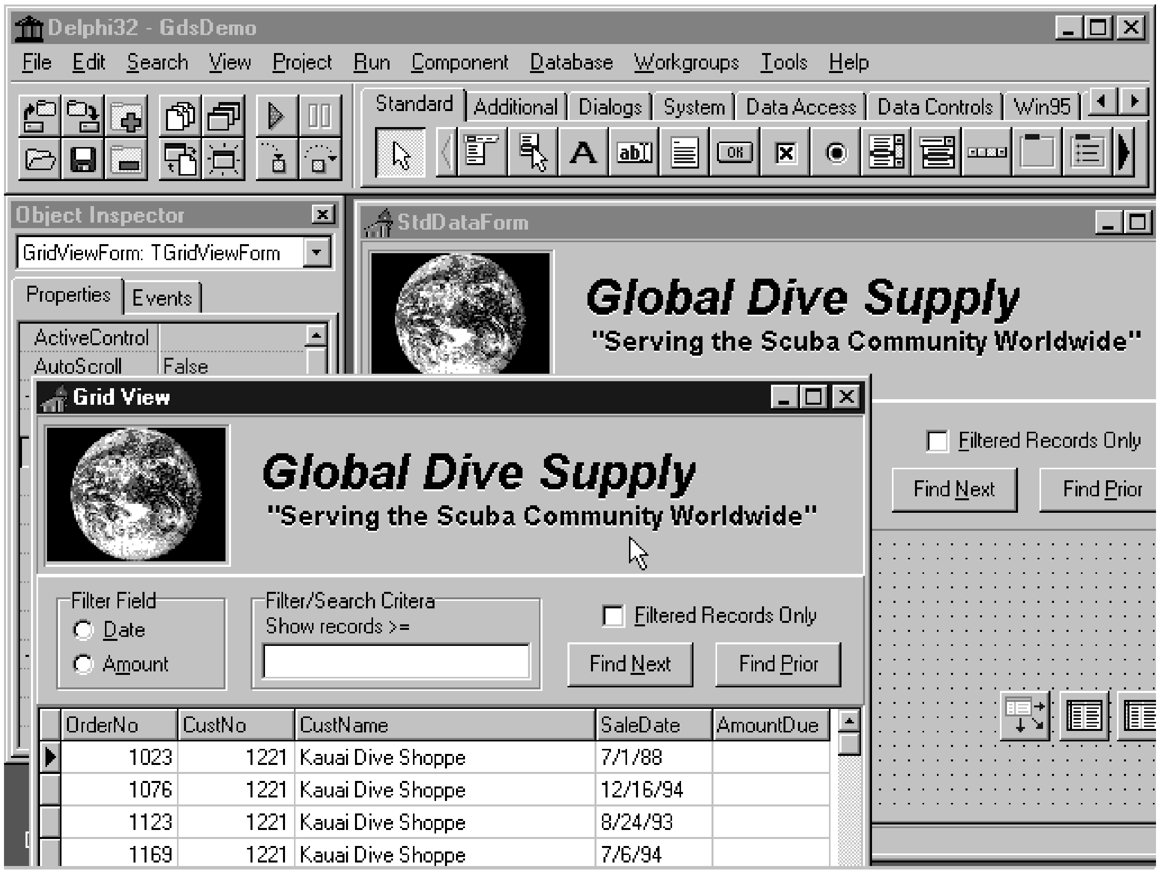
Visual Form Inheritance

Summary: Delphi 2.0 extends the support for OOP by offering new Visual Form Inheritance.

One of Delphi's most important capabilities has been the full support for object-oriented programming. By having full support for encapsulation, polymorphism and inheritance in the Object Pascal language, Delphi has the unique capability of allowing developers to create their own custom objects, whether they subclass from existing visual components in Delphi or represent entirely new abstract business objects. And since Delphi is written in Delphi, there is no distinction between the types of objects a developer or third party could create and those written by Borland. In fact, many third party vendors and Delphi developers have created a large and growing marketplace for these components.

Delphi 2.0 takes the fundamental OOP capabilities and extends them to the visual environment in order to make inheritance easier to use and more accessible. You can now visually inherit from forms while in the design environment, without writing code, and immediately see the effects of changes. For example, in many corporate environments it's desirable to create a standard "template" form, say for data entry, which will be used as the basis for many other forms. By using Visual Form Inheritance, you can be assured that as changes are made to the standard form they can be immediately inherited by other forms.

Visual Form Inheritance allows you to inherit all of the code, objects and properties with as many levels of inheritance as you like without runtime performance penalties. Other systems which attempt to implement a inheritance have severe performance penalties which render the feature unusable in the real world.


Delphi32 - GdsDemo
File Edit Search View Project Run Component Database Workgroups Tools Help
Standard Additional Dialogs System Data Access Data Controls Win95
Object Inspector
Properties Events
StdDataForm
Global Dive Supply
"Serving the Scuba Community Worldwide"
Grid View
Global Dive Supply
"Serving the Scuba Community Worldwide"
Filter Field
Date
Amount
Filter/Search Critera
Show records >=
Filtered Records Only
Find Next
Find Prior

GridViewForm: TGridViewForm

ActiveControl	
AutoScroll	False

OrderNo	CustNo	CustName	SaleDate	AmountDue
1023	1221	Kauai Dive Shoppe	7/1/88	
1076	1221	Kauai Dive Shoppe	12/16/94	
1123	1221	Kauai Dive Shoppe	8/24/93	
1169	1221	Kauai Dive Shoppe	7/6/94	

Figure 12 -- Delphi's Visual Form Inheritance allows you to easily create reusable, shared forms

Visual Form Inheritance Example

The sample application \DEMOS\DB\GDSDEMO\GDSDEMO.DPR contains an example of Visual Form Inheritance. This example includes two forms which are presented to the user. These are the GridViewForm and RecViewForm which display a Grid view and Record view, respectively, on a common data set. These two forms share many common elements and, in fact, inherit those common elements from an ancestor form type, the StdDataForm. The StdDataForm defines the Customer and Orders tables, a DataSource, code for applying a filter on the Orders tables and code for finding the next or previous Order that meets the filter criteria. Because so much of the functionality is defined in the StdDataForm, there is very little code required in the GridViewForm and no code whatsoever in the RecViewForm.

Because of the use of Visual Form Inheritance, any changes made to the StdDataForm, whether in code or visually using the Object Inspector or form designer, are automatically propagated to the descendants GridViewForm and RecViewForm. For example, if you move the Find Next button or change it's code in the StdDataForum you'll immediately see the change in both of the descendent forms. Of course, you are also free to override the visual properties or code of any of the inherited components in the GridViewForm.

Delphi's Visual Form Inheritance uses a property level of granularity in determining what is inherited and what is overridden. This means that if you change, say, the location and Font of the Find Next button in the GridViewForm, it will still inherit all of the other properties including caption, size and the code associated with the OnClick event handler.

If you decide that you would like to revert back to all of the ancestor component's properties, you can right click on a component in the descendant form and select the menu choice Revert to Inherited. This will effectively disable all overrides to properties that you have changed.

Calling Inherited Event Handlers

When you want to add or override behavior of a component in a descendent form you can simply use the object Inspector, as with any Delphi component, to attach code to the particular event handlers. For example, to override the code associated with the OnClick event handler for the Find Next button in the GridViewForm, you can simply double click on the Find Next button or double click on the OnClick property in the Object Inspector. The method that is generated will include a call to the inherited method defined in the ancestor as shown below.

```
procedure TGridViewForm.NextBtnClick(Sender: TObject);
begin
  inherited;
  { Add new code here }
end;
```

Note: For easier code maintenance you should always call the inherited routine. If one does not exist, the call will be ignored.

Creating a New Inherited Form from the Object Repository

To create a new inherited form, use the File New menu command to bring up the Object Repository. You can then select any form in the current project page (e.g. GdsDemo) or from the Forms, Dialogs or Data Modules pages. Note that for forms in the current project the only available option is Inherit, whereas you can also select Copy or Reference when selecting from the Forms, Dialogs or Data Modules pages. For more information on these topics, refer to the on-line Help on the Object Repository.

How Form Inheritance is Implemented

When you inherit from a form, whether from the current project or from the Object Repository, the ancestor class is loaded into memory. If you look at the definition for an inherited form in the Code Editor, you will see that instead of inheriting from TForm, it inherits from some other form type. For example, here's the definition for the GridViewForm:

```
type
  TGridViewForm = class(TStdDataForm)
    DBGrid1: TDBGrid;
    procedure NextBtnEndDrag(Sender, Target: TObject; X, Y: Integer);
    procedure NextBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

You can see that it contains an additional object in it's definition for the DBGrid control; all of the GridViewForm's other components, such as the buttons, tables are defined in the ancestor type, TStdDataForm, shown below.

```
TStdDataForm = class(TGDSStdForm)
    StdCtrlPanel: TPanel;
    FilterOnRadioGroup: TRadioGroup;
    Orders: TTable;
    Cust: TTable;
    OrdersSource: TDataSource;
    GroupBox1: TGroupBox;
    FilterOnLabel: TLabel;
    FilterCriteria: TEdit;
    FilterCheckBox: TCheckBox;
    NextBtn: TButton;
    PriorBtn: TButton;
...
    procedure FilterOnRadioGroupClick(Sender: TObject);
    procedure OrdersCalcFields(DataSet: TDataSet);
    procedure FilterCheckBoxClick(Sender: TObject);
    procedure PriorBtnClick(Sender: TObject);
    procedure NextBtnClick(Sender: TObject);
    procedure FilterCriteriaExit(Sender: TObject);
    procedure FilterCriteriaKeyPress(Sender: TObject; var Key: Char);
  protected
    FLastAmount: Double;
    FLastDate: TDateTime;
    function CalcAmountDue: Double;
    procedure ConvertFilterCriteria;
  end;
```

Similarly, if you right click on the GridViewForm in the form designer and select View as Text you can see exactly which properties have been changed from the ancestor. The example shows the GridViewForm text representation like assuming that the Find Next button has a different font and location.

```
inherited GridViewForm: TGridViewForm
```

```
  Caption = 'Grid View'
  inherited StdCtrlPanel: TPanel
    inherited NextBtn: TButton
      Left = 223
      Top = 13
      Font.Height = -13
      Font.Style = [fsBold]
      ParentFont = False
    end
  end
  object DBGrid1: TDBGrid [2]
    Left = 0
    Top = 161
    Width = 460
    Height = 159
    Align = alClient
    DataSource = OrdersSource
    TabOrder = 2
    TitleFont.Color = clBlack
    TitleFont.Height = -11
    TitleFont.Name = 'MS Sans Serif'
    TitleFont.Style = []
  end
end
```

Visual Form Inheritance is implemented by streaming out the property sets on a component by component basis for the ancestor form and it's descendants and then comparing the differences. Component writers should ensure that their components write out only the properties necessary so that the Visual Form Inheritance difference analysis can be as efficient as possible. Some components use a large granularity when determining which properties have changed in the descendent. For example, the Database Grid control uses a Collection property to store the column attributes. Therefore, if you change any of the properties of such a collection in a descendant form, the entire collection is considered to override the ancestor's collection. This means that if you change the column attributes in a descendant grid in an inherited form, any changes to the column attributes in the ancestor will no longer propagate to the descendant. Hence, the granularity for column attributes is the entire column attribute set, rather than each distinct column. However, changes made to other properties in the ancestor grid, such as the Font or Color will automatically propagate to the descendant, thereby giving you a good balance of flexibility and efficiency.

Note: If you use a deep level of visual form inheritance, you may notice a slight delay in repainting complex forms when you resize or move them in the design environment on slower machines. This performance slow down only occurs in the design environment since Delphi must compute the differences between the property sets of the ancestor and descendant. During the compilation process, all of this information is compiled into a static set of properties and so once the forms are loaded there is no such impact on performance. Both inherited forms and non-inherited forms will behave identically at runtime.

The Advantage of Native Code Compilers

Delphi always compiles immediately to optimized native machine code, unlike 4GL systems which sometimes allow you to generate C code which can be then compiled with a separate C code compiler. Although this sounds like an attractive way of correcting the major performance deficiencies of 4GL systems, in fact, Delphi has shown that there's a more efficient and reliable way to develop applications by always directly generating optimized native machine code. Because Delphi always produces optimized native machine code, it offers a number of advantages over a 2 stage "code generator" approach. These advantages include:

- **faster turnaround --** by having the world's fastest native code compiler, Delphi increases productivity and encourages Rapid Application Development. Unlike 2 stage "C code generators", you never have to wait for the code generator or go through a separate generate / compile / link cycle.
- **easier testing** -- the code you work with in the development environment is the same compiled code you'll deploy in your production application. This means you don't have to worry about whether code that works with the interpreter will behave the same as the code created by the compiler.
- **high level debugging --** since code that you write is compiled directly into machine code, the debugger lets you debug the code you wrote from within the environment. You never have to look at cryptic C code generated by the system. You never have to debug the C code created by the code generator.
- **easiest maintenance --** you only have to maintain code in a single high-level language, object Pascal, rather than having to maintain code in both a 4GL language and in low-level C.
    - **easier deployment** -- you can distribute your applications without runtime interpreter DLLs making it easy to deploy in the field.

Historically P-code systems and code generators have proven to be useful stop gap measures. For example, some of the early language implementations on the Apple II computer were implemented using the UCSD P-code system, which allowed programmers to work with high level languages such as Pascal instead of Basic on machines equipped with as little as 64K. However, since that time, many developers have been reluctant to use P-code systems due to their inherently slower performance.

Similarly, many of the first C++ language implementations were created as source code translators which translated C++ code into low-level C code which could then be compiled with any standard C compiler. However, the disadvantages of a slow turnaround time due to the two-stage compile process and the difficulty of debugging the generated C code, caused most C++ developers to seek "native code" C++ compilers instead.

Compatibility with 16 bit code

Our goal with Delphi 2.0 is to offer full compatibility of 16-bit code. In order to migrate code from the 16-bit version to the 32-bit version it must be recompiled, but otherwise, few changes are necessary. In most cases, developers can simply load their 16 bit applications into the new environment, compile the code for 32-bit and begin adding new 32-bit features. In some cases, conversion of code may be necessary, but only where fundamental assumptions are made that depend on representations that have changed in either the Windows environment or code that is dependent upon the low-level physical implementation of data types that have changed in order to move to 32 bits. Delphi automatically handles changes in the Windows message types (known as message cracking) so that message handling code does not need to be changed even where the Windows message data fields have been rearranged or resized to 32 bits. Code which depends on the physical representation will need to be updated including:

- 16 bit in-line assembler
- Windows API functions which have changed in Win32

- Records or routines that depend on the physical size of integers

In Delphi 2.0, the Integer type is assumed to be 32 bits, rather than 16 bits as it was previously. A new type, SmallInt, is provided for compatibility purposes.

The new 32-bit compiler also has the capability of "unit aliasing" which enables you to use a different symbolic name (or alias) for a unit. This is a useful technique when you want to dramatically change the implementation of a unit in order to take advantage of new 32-bit capabilities. For example, in the 16 bit version of Delphi there were two separate units for WinTypes and WinProcs. These units have been combined into a single unit called Windows in the 32-bit version which contains all of the new 32-bit Windows types, functions and messages. In order to provide complete compatibility, unit aliases are included so that old code compiles as is, without needing to change the "uses" statement. Thus the statement:

```
Uses WinTypes, WinProcs;
```

is aliased to be understood as:

```
Uses Windows;
```

Applications that are written in Delphi 2.0 and do not take advantage of the 32-bit features can be recompiled with the 16 bit version of Delphi for use on Windows 3.1. However, if a program takes advantage of 32-bit features, such as Windows 95 user interface elements, Win32 API functions, the 32-bit flat address space or other new features, it will require modifications in order to compile for 16 bits.

Conclusion

Delphi 2.0 is a totally new product built from the ground up on an optimizing compiler to take advantage of the improved performance of 32-bit platforms including Windows 95 and Windows NT. There are many new compiler optimizations, linker optimizations and new 32-bit data types so that the performance advantage of Delphi applications over p-code interpreters is even greater than before. In addition, the new compiler architecture makes it easier to create correct code and also offers easy access to important system features such as multi-threading, OLE automation and OLE controls (OCXs).

Delphi 2.0 also includes many new components for accessing the Windows 95 user interface elements, new database components for accessing features of the new 32-bit version of the Borland Database Engine as well as several new tools that facilitate client/server development and a more extensive Open Tools API for integrating with third party packages such as CASE tools. Additional whitepapers on these topics will be forthcoming.

Delphi 2.0 will be available in first calendar quarter 1996. Exact packaging and pricing information will be made available at a later time.