Turbo Pascal 5.5®

Object-Oriented Programming Guide
All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.
CONTENTS

Introduction
About this manual .................. 1
Installation ................................ 2
Special Notes .......................... 3
Online help ............................. 4
How to contact Borland .............. 4

Chapter 1  All about OOP
Objects? .................. 8
Inheritance .................. 9
Objects: records that inherit ....... 10
Instances of object types ............ 13
An object's fields .................. 13
Good practice and bad practice .... 13
Methods ............................. 14
Code and data together .......... 16
Defining methods .................. 16
Method scope and the Self parameter .................. 17
Object data fields and method formal parameters .............. 19
Objects exported by units .......... 19
Programming in the active voice .. 22
Encapsulation ..................... 23
Methods: no downside ............ 24
Extending objects .................. 25
Inheriting static methods .......... 27
Virtual methods and polymorphism 29
Early binding vs. late binding ..... 30
Object type compatibility ......... 31
Polymorphic objects .............. 33
Virtual methods .................... 35
  Range checking virtual method calls .................. 37
  Once virtual, always virtual .......... 37
  An example of late binding ......... 38
  Procedure or method? .......... 39
  Object extensibility .......... 46
  Static or virtual methods .......... 48
  Dynamic objects .......... 49
  Allocation and initialization with
    New .................. 50
  Disposing dynamic objects .......... 51
  Destructors .................. 52
  An example of dynamic object allocation .......... 54
    Disposing of a complex data structure on the heap .......... 55
Where to now? .................... 60
Conclusion ..................... 61

Chapter 2 Object-oriented debugging
Object-oriented debugging in the IDE .. 63
Stepping and tracing method calls .......... 63
Objects in the Evaluate window .......... 64
Objects in the Watch window .......... 64
Expressions in the Find Procedure command .......... 64
Turbo Debugger .................. 65
Stepping and tracing method calls .......... 65
Scope .................. 65
Evaluate Window .................. 66
  Calling methods in the Evaluate window .......... 67
Watch window .................... 67
The Object Hierarchy window .......... 67
  The object type list pane .......... 68
  The local menu .......... 68
  The hierarchy tree pane .......... 68
The Object Type Inspector window .......... 69
  The local menus .......... 69
<table>
<thead>
<tr>
<th>Chapter 3 Turbo Pascal 5.5 language definition</th>
<th>73</th>
</tr>
</thead>
<tbody>
<tr>
<td>New reserved words</td>
<td>73</td>
</tr>
<tr>
<td>Object types</td>
<td>73</td>
</tr>
<tr>
<td>Assignment compatibility</td>
<td>78</td>
</tr>
<tr>
<td>Object component designators</td>
<td>78</td>
</tr>
<tr>
<td>Dynamic object type variables</td>
<td>79</td>
</tr>
<tr>
<td>Instance initialization</td>
<td>79</td>
</tr>
<tr>
<td>Object type constants</td>
<td>80</td>
</tr>
<tr>
<td>@ with a method</td>
<td>80</td>
</tr>
<tr>
<td>Function calls</td>
<td>81</td>
</tr>
<tr>
<td>Assignment statements</td>
<td>81</td>
</tr>
<tr>
<td>Procedure statements</td>
<td>81</td>
</tr>
<tr>
<td>Case statements</td>
<td>82</td>
</tr>
<tr>
<td>With statements</td>
<td>82</td>
</tr>
<tr>
<td>Method declarations</td>
<td>83</td>
</tr>
<tr>
<td>Constructors and destructors</td>
<td>84</td>
</tr>
<tr>
<td>Variable parameters</td>
<td>85</td>
</tr>
<tr>
<td>Extensions to New and Dispose</td>
<td>86</td>
</tr>
<tr>
<td>Compiler directive conditional symbols</td>
<td>87</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4 Overlays</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlay buffer management</td>
<td>89</td>
</tr>
<tr>
<td>Variables</td>
<td>91</td>
</tr>
<tr>
<td>OvrTrapCount</td>
<td>91</td>
</tr>
<tr>
<td>OvrLoadCount</td>
<td>92</td>
</tr>
<tr>
<td>OvrFileMode</td>
<td>92</td>
</tr>
<tr>
<td>OvrReadBuf</td>
<td>92</td>
</tr>
<tr>
<td>Procedures and functions</td>
<td>94</td>
</tr>
<tr>
<td>OvrSetRetry</td>
<td>94</td>
</tr>
<tr>
<td>OvrGetRetry</td>
<td>95</td>
</tr>
<tr>
<td>Overlays in .EXE files</td>
<td>95</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5 Inside Turbo Pascal</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal data format of objects</td>
<td>97</td>
</tr>
<tr>
<td>Virtual method tables</td>
<td>98</td>
</tr>
<tr>
<td>The SizeOf standard function</td>
<td>100</td>
</tr>
<tr>
<td>The TypeOf standard function</td>
<td>101</td>
</tr>
<tr>
<td>Virtual method calls</td>
<td>101</td>
</tr>
<tr>
<td>Method calling conventions</td>
<td>102</td>
</tr>
<tr>
<td>Constructors and destructors</td>
<td>102</td>
</tr>
<tr>
<td>Assembly language methods</td>
<td>103</td>
</tr>
<tr>
<td>Constructor error recovery</td>
<td>106</td>
</tr>
</tbody>
</table>

| Appendix A New and modified error messages      | 111 |

| Index                                          | 113 |
FIGURES

1.1: A partial taxonomy chart of insects . . 9
1.2: Layout of program ListDemo's data structures . 55
4.1: Loading and disposing overlays . . . 90
5.1: Layouts of instances of Location, Point, and Circle . . . 98
5.2: Point and Circle’s VMT layouts . . . 100
Turbo Pascal 5.5 gives you the power and efficiency of object-oriented programming at turbo speed. In addition to the Turbo Pascal features you have come to rely on, this new version offers you the programming techniques of the future:

- both static objects for maximum efficiency and dynamic objects for maximum run-time flexibility
- both static and virtual methods
- constructors and destructors that create and deallocate objects (which saves programming time and improves readability of your code)
- object constants—static object data is initialized automatically
- greater speed—Turbo Pascal 5.5 compiles even faster
- an improved overlay manager (which lets overlays run faster, with less disk I/O)
- enhanced help screens that let you cut and paste examples into your code
- an online tutorial to introduce you to Turbo Pascal's integrated development environment

The object-oriented extensions in Turbo Pascal 5.5 were inspired by Larry Tesler's "Object Pascal Report" (Apple, 1985) and Bjarne Stroustrup's "The C++ Programming Language" (1986, Addison-Wesley).

About this manual

This manual contains information on the new object-oriented features of Turbo Pascal 5.5. For all other information about Turbo Pascal, refer to the Turbo Pascal User's Guide or the Turbo Pascal Reference Guide.
Here's a breakdown of the chapters and appendixes in this volume:

- **Chapter 1: All about OOP** introduces you to the main concepts of object-oriented programming—how objects differ from records, the advantages of encapsulated data and code, inheritance, polymorphism, static versus dynamic object instances—and uses practical examples to demonstrate the principles of object-oriented programming.

- **Chapter 2: Object-oriented debugging** covers modifications to Turbo Debugger to support Turbo Pascal 5.5, including Object Inspectors and the Object Hierarchy window.

- **Chapter 3: Turbo Pascal 5.5 language definition** contains the formal definition of all object-oriented extensions to Turbo Pascal.

- **Chapter 4: Overlays** discusses improvements to the Turbo Pascal overlay manager.

- **Chapter 5: Inside Turbo Pascal** explains the implementation of the object-oriented features of Turbo Pascal 5.5.

- **Appendix A: New and modified error messages** lists new compiler error messages and warnings specific to object-oriented Turbo Pascal.

---

**Installation**

The first thing you’ll want to do is install Turbo Pascal on your system. Your Turbo Pascal package includes all the files and programs necessary to run both the integrated environment and command-line versions of the compiler. The INSTALL program sets up Turbo Pascal on your system, and it works on both hard-disk and floppy-based systems.

INSTALL walks you through the installation process. All you have to do is follow the instructions that appear onscreen at each step. *Please read them carefully.* If you're installing onto floppies, rather than onto a hard disk, be sure to have at least four blank, formatted 360K disks on hand.

To run INSTALL:

1. Insert the distribution disk labeled Installation Disk in Drive A.
2. Type A: and press Enter.
3. Type INSTALL and press Enter.

From this point on, just follow the instructions that INSTALL displays onscreen.

As soon as INSTALL is finished running, you are ready to start using Turbo Pascal.

After you’ve tried out the Turbo Pascal integrated development environment, you may want to customize some of the options. To do that, use the program TINST, which is discussed in Appendix D of the User’s Guide.

Special Notes

- If you use INSTALL’s Upgrade option, version 5.5 files will overwrite any version 5.0 files that have the same names.
- If you install the graphics files into a separate subdirectory (C:\TP\BGI, for example), remember to specify the full path to the driver and font files when you call InitGraph. For example,
  \[\text{InitGraph(Driver, Mode, 'C:\TP\BGI');}\]
- If GRAPH.TPU is not in the current directory, you’ll need to add its location to the unit directories with the Options/Directories/Unit Directories command (or with the /U option in the command-line compiler) in order to compile a BGI program.
- If you have difficulty reading the text displayed by the INSTALL or TINST programs, they will both accept an optional command-line parameter that forces them to use black-and-white colors:
  - A:INSTALL /B Forces INSTALL into BW80 mode
  - A:TINST /B Forces TINST into BW80 mode

You may need to specify the /B parameter if you are using an LCD screen or a system that has a color graphics adapter and a monochrome or composite monitor. To find out how to permanently force the integrated environment to use black-and-white colors with your LCD screen (or CGA and monochrome/composite monitor combination), see the note on page 26 of the User’s Guide.
Online help

You can get online help about both the integrated environment and language-specific items. To bring up help when you're on a menu item or within a window, press F1; to bring up the main index to the help system, press F1 again.

Language-specific help is available only when you're in the Edit window by pressing Ctrl-F1. You can get help about the syntax of Pascal reserved words or the usage and parameters of a particular procedure or function, cut and paste examples into your file, or find out about compiler switches, and more.

For language help, position your cursor on the item in the Edit window you want to know more about and then press Ctrl-F1.

To cut and paste from help, follow these easy steps:

1. Once you've brought up the help screen you want to copy from, press C. This activates the cursor so you can position it anywhere on the help screen.

2. After you've placed the cursor at the beginning of the text you want to copy, press B to begin. Then use the ↑, ↓, →, and ← arrow keys to move to the end of your block (highlighting the text you're copying at the same time). Pressing B again resets the beginning of the block to the cursor position.

3. To end cut-and-paste and to place the text in your edit file, press Enter.

4. The text is pasted into the editor and is marked as a block, which allows you to easily move the pasted block.

How to contact Borland

If, after reading this manual and using Turbo Pascal, you'd like to contact Borland with comments for technical support, we suggest the following procedures:

- The best way is to log on to Borland's forum on CompuServe: Type GO BPROGA at the main CompuServe menu and follow the menus to section 2. Leave your questions or comments here for the support staff to process.

- If you prefer, write a letter and send it to
If you include a program example in your letter, it must be limited to 100 lines or less. We request that you submit it on disk, include all the necessary support files on that disk, and provide step-by-step instructions on how to reproduce the problem. Before you decide to get technical support, try to duplicate the problem with the code contained on the floppy disk, just to be sure we can duplicate the problem using the disk you provide us.

You can also telephone our Technical Support department at (408) 438-5300. To help us handle your problem as quickly as possible, have these items handy before you call:

- product name and version number
- product serial number
- computer make and model number
- operating system and version number
Object-oriented programming is a method of programming that closely mimics the way all of us get things done. It is a natural evolution from earlier innovations to programming language design: It is more structured than previous attempts at structured programming; and it is more modular and abstract than previous attempts at data abstraction and detail hiding. Three main properties characterize an object-oriented programming language:

- **Encapsulation**: Combining a record with the procedures and functions that manipulate it to form a new data type: an object.
- **Inheritance**: Defining an object and then using it to build a hierarchy of descendant objects, with each descendant inheriting access to all its ancestors’ code and data.
- **Polymorphism**: Giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.

Turbo Pascal 5.5’s language extensions give you the full power of object-oriented programming: more structure and modularity, more abstraction, and reusability built right into the language. All these features add up to code that is more structured, extensible, and easy to maintain.

The challenge of object-oriented programming (OOP) is that it sometimes requires you to set aside habits and ways of thinking about programming that have been considered standard for many years. Once that is done, however, OOP is simple, straight-
forward, and superior for solving many of the problems that plague traditional software programs.

A note to you who have done object-oriented programming in other languages: Put aside your previous impressions of OOP and learn Turbo Pascal 5.5's object-oriented features on their own terms. OOP is not one single way; it is a continuum of ideas. In its object philosophy, Turbo Pascal 5.5 is more like C++ than Smalltalk. Smalltalk is an interpreter, while from the beginning, Turbo Pascal has been a pure native code compiler. Native code compilers do things differently (and far more quickly) than interpreters. Turbo Pascal was designed to be a production development tool, not a research tool.

And a note to you who haven't any notion at all what OOP is about: That's just as well. Too much hype, too much confusion, and too many people talking about something they don't understand have greatly muddied the waters in the last year or so. Strive to forget what people have told you about OOP. The best way (in fact, the only way) to learn anything useful about OOP is to do what you're about to do: Sit down and try it yourself.

Objects?

Yes, objects. Look around you...there's one: the apple you brought in for lunch. Suppose you were going to describe an apple in software terms. The first thing you might be tempted to do is pull it apart: Let $S$ represent the area of the skin; let $J$ represent the fluid volume of juice it contains; let $F$ represent the weight of fruit inside; let $D$ represent the number of seeds....

Don't think that way. Think like a painter. You see an apple, and you paint an apple. The picture of an apple is not an apple; it's just a symbol on a flat surface. But it hasn't been abstracted into seven numbers, all standing alone and independent in a data segment somewhere. Its components remain together, in their essential relationships to one another.

Objects model the characteristics and behavior of the elements of the world we live in. They are the ultimate data abstraction (so far).

An apple can be pulled apart, but once it's been pulled apart it's not an apple anymore. The relationships of the parts to the whole and to one another are plainer when everything is kept together.
in one wrapper. This is called encapsulation, and it's very important. We'll return to encapsulation in a little while.

Of equal importance is the fact that objects can inherit characteristics and behavior from what we call ancestor objects. This is an intuitive leap; inheritance is perhaps the single biggest difference between object-oriented Pascal and Turbo Pascal programming today.

**Inheritance**

The goal of science is to describe the workings of the universe. Much of the work of science, in furthering that goal, is simply the creation of family trees. When entomologists return from the Amazon with a previously unknown insect in a jar, their fundamental concern is working out where that insect fits into the giant chart upon which the scientific names of all other insects are gathered. There are similar charts of plants, fish, mammals, reptiles, chemical elements, subatomic particles, and external galaxies. They all look like family trees: a single overall category at the top, with an increasing number of categories beneath that single category, fanning out to the limits of diversity.

Within the category insect, for example, there are two divisions: insects with visible wings, and insects with hidden wings or no wings at all. Under winged insects is a larger number of categories: moths, butterflies, flies, and so on. Each category has numerous subcategories, and beneath those subcategories are even more subcategories (see Figure 1.1).

![A partial taxonomy chart of insects](image)
This classification process is called taxonomy. It’s a good starting metaphor for the inheritance mechanism of object-oriented programming.

The questions that a scientist asks in trying to classify some new animal or object are these: How is it similar to the others of its general class? How is it different? Each different class has a set of behaviors and characteristics that define it. A scientist begins at the top of a specimen’s family tree and starts descending the branches, asking those questions along the way. The highest levels are the most general, and the questions the simplest: Wings or no wings? Each level is more specific than the one before it, and less general. Eventually the scientist gets to the point of counting hairs on the third segment of the insect’s hind legs—specific indeed. (And a good reason, perhaps, not to be an entomologist.)

The important point to remember is that once a characteristic is defined, all the categories beneath that definition include that characteristic. So once you identify an insect as a member of the order diptera (flies), you needn’t make the point again that a fly has one pair of wings. The species of insect we call flies inherits that characteristic from its order.

As you’ll learn shortly, object-oriented programming is very much the process of building family trees for data structures. One of the important things object-oriented programming adds to traditional languages like Pascal is a mechanism for data types to inherit characteristics from simpler, more general types. This mechanism is inheritance.

**Objects: records that inherit**

In Pascal terms, an object is very much like a record, which is a wrapper for joining several related elements of data together under one name. In a graphics environment, we might gather together the X and Y coordinates of a position on the graphics screen and call it a record type named *Location*:

```pascal
Location = record
  X, Y : Integer;
end;
```

*Location* here is a record type; that is, it’s a template that the compiler uses to create record variables. A variable of type *Location* is an instance of type *Location*. The term *instance* is used now and
then in Pascal circles, but it is used all the time by OOP people, and you’ll do well to start thinking in terms of types and instances of those types.

With type Location you have it both ways: When you need to think of the X and Y coordinates separately, you can think of them separately as fields X and Y of the record. On the other hand, when you need to think of the X and Y coordinates working together to pin down a place on the screen, you can think of them collectively as Location.

Suppose you wanted to display a point of light at a position described on the screen by a Location record. In Pascal you might add a Boolean field indicating whether there is an illuminated pixel at a given location, and make it a new record type:

```
Point = record
  X, Y : Integer;
  Visible : Boolean;
end;
```

You might also be a little more clever and retain record type Location by creating a field of type Location within type Point:

```
Point = record
  Position : Location;
  Visible : Boolean;
end;
```

This works, and Pascal programmers do it all the time. One thing this method doesn’t do is force you to think about the nature of what you’re manipulating in your software. You need to ask questions like, “How does a point on the screen differ from a location on the screen?” The answer is this: A point is a location that lights up. Think back on the first part of that statement: A point is a location.... There you have it!

Implicit in the definition of a point is a location for that point. (Pixels exist only on the screen, after all.) In object-oriented programming, we recognize that special relationship. Because all points must contain a location, we say that type Point is a descendant type of type Location. Point inherits everything that Location has, and adds whatever is new about Point to make Point what it must be.

This process by which one type inherits the characteristics of another type is called inheritance. The inheritor is called a descen-


*dant type*; the type that the descendant type inherits from is an *ancestor type*.

The familiar Pascal record types cannot inherit. Turbo Pascal 5.5, however, extends the Pascal language to support inheritance. One of these extensions is a new category of data structure, related to records but far more powerful. Data types in this new category are defined with a new reserved word: *object*. An object type can be defined as a complete, stand-alone type in the fashion of Pascal records, or it can be defined as a descendant of an existing object type, by placing the name of the ancestor type in parentheses after the reserved word *object*.

In the graphics example you just looked at, the two related object types would be defined this way:

```pascal
type
  Location = object
    X, Y : Integer;
  end;

  Point = object(Location)
    Visible : Boolean;
  end;
```

Here, *Location* is the ancestor type, and *Point* is the descendant type. As you’ll see a little later, the process can continue indefinitely: You can define descendants of type *Point*, and descendants of *Point*’s descendant type, and so on. A large part of designing an object-oriented application lies in building this *object hierarchy* expressing the family tree of the objects in the application.

All the eventual types inheriting from *Location* are called *Location*’s descendant types, but *Point* is one of *Location*’s immediate descendants. Conversely, *Location* is *Point*’s immediate ancestor. An object type (just like a DOS subdirectory) can have any number of immediate descendants, but only one immediate ancestor.

Objects are closely related to records, as these definitions show. The new reserved word *object* is the most obvious difference, but there are numerous other differences, some of them quite subtle, as you’ll see later.

For example, the X and Y fields of *Location* are not explicitly written into type *Point*, but *Point* has them anyway, by virtue of inher-
Instances of object types

Instances of object types are declared just as any variables are declared in Pascal, either as static variables or as pointer referents allocated on the heap:

```pascal
type
  PointPtr = ^Point;

var
  StatPoint : Point;   { Ready to go! }
  DynaPoint : PointPtr;  { Must allocate with New before use }
```

An object's fields

You access an object's data fields just as you access the fields of an ordinary record, either through the `with` statement or by `dotting`. For example,

```pascal
MyPoint.Visible := False;

with MyPoint do
begin
  X := 341;
  Y := 42;
end;
```

You will just have to remember at first (it will eventually come naturally) that inherited fields are just as accessible as fields declared within a given object type. For example, even though `X` and `Y` are not part of `Point`'s declaration (they are inherited from type `Location`), you can specify them just as though they were declared within `Point`:

```pascal
MyPoint.X := 17;
```

Don't forget: An object's inherited fields are not treated specially simply because they are inherited.

Good practice and bad practice

Even though you can access an object's fields directly, it's not an especially good idea to do so. Object-oriented programming principles require that an object's fields be left alone as much as possible. This restriction might seem arbitrary and rigid at first, but it's part of the big picture of OOP that we're building in this chapter. In time you'll see the sense behind this new definition of good programming practice, though there's some ground to cover.
An object's data fields are what an object knows; its methods are what an object does.

before it all comes together. For now, take it on faith: Avoid accessing object data fields directly.

So—how are object fields accessed? What sets them and reads them?

The answer is that an object's *methods* should be used to access an object's data fields whenever possible. A *method* is a procedure or function declared *within* an object and tightly bonded to that object.

Methods

Methods are one of object-oriented programming's most striking attributes, and they take some getting used to. Start by harkening back to that fond old necessity of structured programming, initializing data structures. Consider the task of initializing a record with this definition:

```pascal
Location = record
  X, Y : Integer;
end;
```

Most programmers would use a *with* statement to assign initial values to the X and Y fields:

```pascal
var
  MyLocation : Location;
with MyLocation do
begin
  X := 17;
  Y := 42;
end;
```

This works well, but it's tightly bound to one specific record instance, *MyLocation*. If more than one *Location* record needs to be initialized, you'll need more *with* statements that do essentially the same thing. The natural next step is to build an initialization procedure that generalizes the *with* statement to encompass any instance of a *Location* type passed as a parameter:

```pascal
procedure InitLocation(var Target : Location;
                       NewX, NewY : Integer);
begin
  with Target do
  begin
```
X := NewX;
Y := NewY;
end;
end;

This does the job, all right—but if you're getting the feeling that it's a little more fooling around than it ought to be, you're feeling the same thing that object-oriented programming's early proponents felt.

It's a feeling that implies that, well, you've designed procedure InitLocation specifically to serve type Location. Why, then, must you keep specifying what record type and instance InitLocation acts upon? There should be some way of welding together the record type and the code that serves it into one seamless whole.

Now there is. It's called a method. A method is a procedure or function welded so tightly to a given type that the method is surrounded by an invisible with statement, making instances of that type accessible from within the method. The type definition includes the header of the method. The full definition of the method is qualified with the name of the type. Object type and object method are the two faces of this new species of structure called an object:

```pascal
type
  Location = object
    X, Y : Integer;
    procedure Init(NewX, NewY : Integer);
  end;

procedure Location.Init(NewX, NewY : Integer);
begin
  X := NewX; { The X field of a Location object }
  Y := NewY; { The Y field of a Location object }
end;
```

Now, to initialize an instance of type Location, you simply call its method as though the method were a field of a record, which in one very real sense it is:

```pascal
var
  MyLocation : Location;

MyLocation.Init(17, 42); { Easy, no? }
```
Code and data together

One of the most important tenets of object-oriented programming is that the programmer should think of code and data together during program design. Neither code nor data exists in a vacuum. Data directs the flow of code, and code manipulates the shape and values of data.

When your data and code are separate entities, there's always the danger of calling the right procedure with the wrong data or the wrong procedure with the right data. Matching the two is the programmer's job, and while Pascal's strong typing does help, at best it can only say what doesn't go together.

Pascal says nothing, anywhere, about what does go together. If it's not in a comment or in your head, you take your chances.

By bundling code and data declarations together, an object helps keep them in sync. Typically, to get the value of one of an object's fields, you call a method belonging to that object that returns the value of the desired field. To set the value of a field, you call a method that assigns a new value to that field.

Turbo Pascal 5.5 does not enforce this, however. Like structured programming, object-oriented programming is a discipline you must enforce upon yourself, using tools provided by the language. Turbo Pascal allows you to read and write an object's fields directly from outside the object—but it encourages you to follow good OOP practice and create methods to manipulate an object's fields from within the object.

Defining methods

The process of defining an object's methods is reminiscent of Turbo Pascal units. Inside an object, a method is defined by the header of the function or procedure acting as a method:

```pascal
 type
 Location = object
   X, Y : Integer;
     procedure Init(InitX, InitY : Integer);
     function GetX : Integer;
     function GetY : Integer;
 end;
```
All data fields must be declared before the first method declaration.

As with procedure and function declarations in a unit’s interface section, method declarations within an object tell what a method does, but not how.

The how is defined outside the object definition, in a separate procedure or function declaration. When methods are fully defined outside the object, the method name must be preceded by the name of the object type that owns the method, followed by a period:

```pascal
procedure Location.Init(InitX, InitY : Integer);
begin
  X := InitX;
  Y := InitY;
end;

function Location.GetX : Integer;
begin
  GetX := X;
end;

function Location.GetY : Integer;
begin
  GetY := Y;
end;
```

Method definition follows the intuitive dotting method of specifying a field of a record. In addition to having a definition of `Location.GetX`, it would be completely legal to define a procedure named `GetX` without the identifier `Location` preceding it. However, the “outside” `GetX` would have no connection to the object type `Location` and would probably confuse the sense of the program as well.

Notice that nowhere in the previous methods is there an explicit `with object do...` construct. The data fields of an object are freely available to that object’s methods. Although separated in the source code, the method bodies and the object’s data fields really share the same scope.

This is why one of `Location`’s methods can contain the statement `GetY := Y` without any qualifier to `Y`. It’s because `Y` belongs to the object that called the method. When an object calls a method, there is an implicit statement to the effect `with myself do` method linking the object and its method in scope.

---

**Chapter 1, All about OOP**
This example is not fully correct syntactically: it's here simply to give you a fuller appreciation for the special link between an object and its methods.

Explicit use of Self is legal, but you should avoid situations that require it.

Methods implemented as externals in assembly language must take Self into account when they access method parameters on the stack. For more details on method call stack frames, see page 102.

This implicit with statement is accomplished by the passing of an invisible parameter to the method each time any method is called. This parameter is called Self, and is actually a full 32-bit pointer to the object instance making the method call. The GetY method belonging to Location is roughly equivalent to the following:

```
function Location.GetY(var Self: Location) : Integer;
begin
  GetY := Self.Y;
end;
```

Is it important for you to be aware of Self? Ordinarily, no. Turbo Pascal's generated code handles it all automatically in virtually all cases. There are a few circumstances, however, when you might have to intervene inside a method and make explicit use of the Self parameter.

Self is actually an automatically declared identifier, and if you happen to find yourself in the midst of an identifier conflict within a method, you can resolve it by using the Self identifier as a qualifier to any data field belonging to the method's object:

```
type
  MouseStat = record
    Active: Boolean;
    X, Y: Integer;
    LButton, RButton: Boolean;
    Visible: Boolean;
  end;

procedure Location.GoToMouse(MousePos: MouseStat);
begin
  Hide;
  with MousePos do
  begin
    Self.X := X;
    Self.Y := Y;
  end;
  Show;
end;
```

This example is necessarily simple, and the use of Self could be avoided simply by abandoning the use of the with statement inside Location.GoToMouse. You might find yourself in a situation inside a complex method where the use of with statements simplifies the logic enough to make Self worthwhile. The Self parameter is part of the physical stack frame for all method calls.
One consequence of the fact that methods and their objects share the same scope is that a method's formal parameters cannot be identical to any of the object's data fields. This is not some new restriction imposed by object-oriented programming, but rather the same old scoping rules that Pascal has always had. It's the same as not allowing the formal parameters of a procedure to be identical to the procedure's local variables:

```pascal
procedure CrunchIt(Crunchee : MyDataRec,
                    Crunchby, ErrorCode : Integer);
var
  A, B : Char;
  ErrorCode : Integer; { This declaration will cause an error! }
begin
  ...

A procedure's local variables and its formal parameters share the same scope and thus cannot be identical. You'll get "Error 4: Duplicate identifier" if you try to compile something like this; the same error occurs if you attempt to give a method a formal parameter identical to any field in the object that owns the method.

The circumstances are a little different, since having procedure headers inside a data structure is a wrinkle new to Turbo Pascal 5.5, but the guiding principles of Pascal scoping have not changed at all.

It makes good sense to define objects in units, with the object type declaration in the interface section of the unit and the procedure bodies of the object type's methods defined in the implementation section of the unit. No special syntactic considerations are necessary to define objects within a unit.

Units can have their own private object type definitions within the implementation section, and such types are subject to the same restrictions as any types defined in a unit implementation section. An object type defined in the interface section of a unit can have descendant object types defined in the implementation section of the unit. In a case where unit B uses unit A, unit B can also define descendant types of any object type exported by unit A.
The object types and methods described earlier can be defined within a unit in this way:

```pascal
unit Points;

interface
uses Graph;

type
  Location = object
    X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    functionGetX : Integer;
    functionGetY : Integer;
  end;

  Point = object(Location)
    Visible : Boolean;
    procedure Init(InitX, InitY : Integer);
    procedure Show;
    procedure Hide;
    function IsVisible : Boolean;
    procedure MoveTo(NewX, NewY : Integer);
  end;

implementation

{--------------------------------------------------------I
  Location's method implementations:
  {--------------------------------------------------------I

procedure Location.Init(InitX, InitY : Integer);
  begin
    X := InitX;
    Y := InitY;
  end;

function Location.GetX : Integer;
  begin
   GetX := X;
  end;

function Location.GetY : Integer;
  begin
   GetY := Y;
  end;

{--------------------------------------------------------I
  Points's method implementations:
  {--------------------------------------------------------I

procedure Point.Init(InitX, InitY : Integer);
```
begin
  Location.Init(InitX, InitY);
  Visible := False;
end;

procedure Point.Show;
begin
  Visible := True;
  PutPixel(X, Y, GetColor);
end;

procedure Point.Hide;
begin
  Visible := False;
  PutPixel(X, Y, GetBkColor);
end;

function Point.IsVisible : Boolean;
begin
  IsVisible := Visible;
end;

procedure Point.MoveTo(NewX, NewY : Integer);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;
end.

To make use of the object types and methods defined in unit Points, you simply use the unit in your own program, and declare an instance of type Point in the var section of your program:

program MakePoints;
uses Graph, Points;

var
  APoint : Point;
  ...

To create and show the point represented by APoint, you simply call APoint's methods using the dot syntax:

APoint.Init(151, 82);  { Initial X,Y at 151,82 }
APoint.Show;           { APoint turns itself on }
APoint.MoveTo(163, 101); { APoint moves to 163,101 }
APoint.Hide;           { APoint turns itself off }
Objects can also be typed constants; see page 80.

Objects, being very similar to records, can also be used inside `with` statements. In that case, naming the object that owns the method isn’t necessary:

```pascal
with APoint do
begin
  Init(151, 82);     { Initial X,Y at 151,82 }
  Show;             { APoint turns itself on }
  MoveTo(163, 101); { APoint moves to 163,101 }
  Hide;             { APoint turns itself off }
end;
```

Just as with records, objects can be passed to procedures as parameters and (as you’ll see later on) be allocated on the heap.

---

Object-oriented languages were once called “actor languages” with this metaphor in mind.

Programming in the active voice

Most of what’s been said about objects so far has been from a comfortable, Turbo Pascal-ish perspective, since that’s most likely where you are coming from. This is about to change, as we move into OOP concepts with fewer precedents in standard Pascal programming. Object-oriented programming has its own particular mindset, due in part to OOP’s origins in the (somewhat insular) research community, but also simply because the concept is truly and radically different.

One often amusing outgrowth of this is that OOP fanatics anthropomorphize their objects. Data structures are no longer passive buckets for you to toss values in. In the new view of things, an object is looked upon as an actor on a stage, with a set of lines (methods) memorized. When you (the director) give the word, the actor recites from the script.

It can be helpful to think of the statement `APoint.MoveTo(242,118)` as giving an order to object `APoint`, saying “Move yourself to location 242,118.” The object is the central concept here. Both the list of methods and the list of data fields contained by the object serve the object. Neither code nor data is boss.

Objects aren’t being described as actors on a stage just to be cute. The object-oriented programming paradigm tries very hard to model the components of a problem as components, and not as logical abstractions. The odds and ends that fill our lives, from toasters to telephones to terry towels, all have characteristics (data) and behaviors (methods). A toaster’s characteristics might include the voltage it requires, the number of slices it can toast at
once, the setting of the light/dark lever, its color, its brand, and so on. Its behaviors include accepting slices of bread, toasting slices of bread, and popping toasted slices back up again.

If we wanted to write a kitchen simulation program, what better way to do it than to model the various appliances as objects, with their characteristics and behaviors encoded into data fields and methods? It's been done, in fact; the very first object-oriented language (Simula-67) was created as a language for writing such simulations.

This is the reason that object-oriented programming is so firmly linked in conventional wisdom to graphics-oriented environments. Objects should be simulations, and what better way to simulate an object than to draw a picture of it? Objects in Turbo Pascal 5.5 should model components of the problem you're trying to solve. Keep that in mind as you further explore Turbo Pascal's new object-oriented extensions.

---

**Encapsulation**

The welding of code and data together into objects is called *encapsulation*. If you're thorough, you can provide enough methods so that a user of the object never has to access its fields directly. Some other object-oriented languages like Smalltalk enforce encapsulation, but in Turbo Pascal 5.5 you have the choice, and good object-oriented programming practice is very much your responsibility.

*Location* and *Point* are written such that it is completely unnecessary to access any of their internal data fields directly:

```pascal
type
  Location = object
    X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    function GetX : Integer;
    function GetY : Integer;
  end;

  Point = object(Location)
    Visible : Boolean;
    procedure Init(InitX, InitY : Integer);
    procedure Show;
    procedure Hide;
    function IsVisible : Boolean;
    procedure MoveTo(NewX, NewY : Integer);
  end;
```

---

*Chapter 1, All about OOP*
There are only three data fields here: X, Y, and Visible. The MoveTo method loads new values into X and Y, and the GetX and GetY methods return the values of X and Y. This leaves no further need to access X or Y directly. Show and Hide toggle the Boolean Visible between True and False, and the IsVisible function returns Visible’s current state.

Assuming an instance of type Point called APoint, you would use this suite of methods to manipulate APoint’s data fields indirectly, like this:

```pascal
with APoint do
begin
  Init(0, 0);  (* Init new point at 0,0 *)
  Show;       (* Make the point visible *)
end;
```

Note that the object’s fields are not accessed at all except by the object’s methods.

---

Adding these methods bulks up Point a little in source form, but the Turbo Pascal smart linker strips out any method code that is never called in a program. You therefore shouldn’t hang back from giving an object type a method that might or might not be used in every program that uses the object type. Unused methods cost you nothing in performance or .EXE file size—if they’re not used, they’re simply not there.

There are powerful advantages to being able to completely decouple Point from global references. If nothing outside the object “knows” the representation of its internal data, the programmer who controls the object can alter the details of the internal data representation—as long as the method headers remain the same.

Within some object, data might be represented as an array, but later on (perhaps as the scope of the application grows and its data volume expands), a binary tree might be recognized as a more efficient representation. If the object is completely encapsulated, a change in data representation from an array to a binary tree will not alter the object’s use at all. The interface to the object remains completely the same, allowing the programmer to fine-tune an object’s performance without breaking any code that uses the object.
People who first encounter Pascal often take for granted the flexibility of the standard procedure *WriteLn*, which allows a single procedure to handle parameters of many different types:

```
WriteLn(CharVar);          { Outputs a character value }
WriteLn(IntegerVar);      { Outputs an integer value }
WriteLn(RealVar);         { Outputs a floating-point value }
```

Unfortunately, standard Pascal has no provision for letting you create equally flexible procedures of your own.

Object-oriented programming solves this problem through inheritance: When a descendant type is defined, the methods of the ancestor type are inherited, but they can also be overridden if desired. To override an inherited method, simply define a new method with the same name as the inherited method, but with a different body and (if necessary) a different set of parameters.

A simple example should make both the process and the implications clear. Let's define a descendant type to *Point* that draws a circle instead of a point on the screen:

```
type
  Circle = object(Point)
    Radius : Integer;
    procedure Init(InitX, InitY : Integer; InitRadius : Integer);
    procedure Show;
    procedure Hide;
    procedure Expand(ExpandBy : Integer);
    procedure MoveTo(NewX, NewY : Integer);
    procedure Contract(ContractBy : Integer);
  end;

procedure Circle.Init(InitX, InitY : Integer; InitRadius : Integer);
begin
  Point.Init(InitX, InitY);
  Radius := InitRadius;
end;

procedure Circle.Show;
begin
  Visible := True;
  Graph.Circle(X, Y, Radius);
end;
```
procedure Circle.Hide;
var
  TempColor : Word;
begin
  TempColor := Graph.GetColor;
  Graph.SetColor(GetBkColor);
  Visible := False;
  Graph.Circle(X, Y, Radius);
  Graph.SetColor(TempColor);
end;

procedure Circle.Expand(ExpandBy : Integer);
begin
  Hide;
  Radius := Radius + ExpandBy;
  if Radius < 0 then Radius := 0;
  Show;
end;

procedure Circle.Contrac t(ContractBy : Integer);
begin
  Expand(-ContractBy);
end;

procedure Circle.MoveTo(NewX, NewY : Integer);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;

A circle, in a sense, is a fat point: It has everything a point has (an X,Y location, a visible/invisible state) plus a radius. Object type Circle appears to have only the single field Radius, but don't forget about all the fields that Circle inherits by being a descendant type of Point. Circle has X, Y, and Visible as well, even if you don't see them in the type definition for Circle.

Since Circle defines a new field, Radius, initializing it requires a new Init method that initializes Radius as well as the inherited fields. Rather than directly assigning values to inherited fields like X, Y and Visible, why not reuse Point's initialization method (illustrated by Circle.Init's first statement). The syntax for calling an inherited method is Ancestor.Method, where Ancestor is the type identifier of an ancestral object type and Method is a method identifier of that type.

Note that calling the method you override is not merely good style; it's entirely possible that Point.Init (or Location.Init for that
matter) performs some important, hidden initialization. By calling the overridden method, you ensure that the descendant object type includes its ancestor's functionality. In addition, any changes made to the ancestor's method automatically affects all its descendants.

After calling `Point.Init`, `Circle.Init` can then perform its own initialization, which in this case consists only of assigning `Radius` the value passed in `InitRadius`.

Instead of drawing and hiding your circle point by point, you can make use of the BGI `Circle` procedure. If you do, `Circle` will also need new `Show` and `Hide` methods that override those of `Point`. These rewritten `Show` and `Hide` methods appear in the example on page 25.

Dotting resolves the potential problems stemming from the name of the object type being the same as that of the BGI routine that draws the object type on the screen. `Graph.Circle` is a completely unambiguous way of telling Turbo Pascal that you're referencing the `Circle` routine in `GRAPH.TPU` and not the `Circle` object type.

Whereas methods can be overridden, data fields cannot. Once you define a data field in an object hierarchy, no descendant type can define a data field with precisely the same identifier.

---

**Inheriting static methods**

One additional `Point` method is overridden in the earlier definition of `Circle: MoveTo`. If you're sharp, you might be looking at `MoveTo` and wondering why `MoveTo` doesn't use the `Radius` field, and why it doesn't make any BGI or other library calls specific to drawing circles. After all, the `GetX` and `GetY` methods are inherited all the way from `Location` without modification. Also, `Circle.MoveTo` is completely identical to `Point.MoveTo`. Nothing was changed other than to copy the routine and give it `Circle`'s qualifier in front of the `MoveTo` identifier.

This example demonstrates a problem with objects and methods set up in this fashion. All the methods shown so far in connection with the `Location`, `Point`, and `Circle` object types are static methods. (The term `static` was chosen to describe methods that are not `virtual`, a term we will introduce shortly. Virtual methods are in fact the solution to this problem, but in order to understand the solution you must first understand the problem.)
The symptoms of the problem are these: Unless a copy of the *MoveTo* method is placed in *Circle's* scope to override *Point's MoveTo*, the method will not work correctly when it is called from an object of type *Circle*. If *Circle* invokes *Point's MoveTo* method, what is moved on the screen is a point rather than a circle. Only when *Circle* calls a copy of the *MoveTo* method defined in its own scope will circles be hidden and drawn by the nested calls to *Show* and *Hide*.

Why so? It has to do with the way the compiler resolves method calls. When the compiler compiles *Point's* methods, it first encounters *Point.Show* and *Point.Hide* and compiles code for both into the code segment. A little later down the file it encounters *Point.MoveTo*, which calls both *Point.Show* and *Point.Hide*. As with any procedure call, the compiler replaces the source code references to *Point.Show* and *Point.Hide* with the addresses of their generated code in the code segment. Thus, when the code for *Point.MoveTo* is called, it in turn calls the code for *Point.Show* and *Point.Hide* and everything's in phase.

So far, this scenario is all classic Turbo Pascal, and would have been true (except for the nomenclature) since version 1.0. Things change, however, when you get into inheritance. When *Circle* inherits a method from *Point*, *Circle* uses the method exactly as it was compiled.

Look again at what *Circle* would inherit if it inherited *Point.MoveTo*:

```pascal
procedure Point.MoveTo(NewX, NewY : Integer);
begin
  Hide;  (* Calls Point.Hide *)
  X := NewX;
  Y := NewY;
  Show;  (* Calls Point.Show *)
end;
```

The comments were added to drive home the fact that when *Circle* calls *Point.MoveTo*, it also calls *Point.Show* and *Point.Hide*, not *Circle.Show* and *Circle.Hide*. *Point.Show* draws a point, not a circle. As long as *Point.MoveTo* calls *Point.Show* and *Point.Hide*, *Point.MoveTo* can't be inherited. Instead, it must be overridden by a second copy of itself that calls the copies of *Show* and *Hide* defined within its scope; that is, *Circle.Show* and *Circle.Hide*.

The compiler's logic in resolving method calls works like this: When a method is called, the compiler first looks for a method of
that name defined within the object type. The Circle type defines methods named Init, Show, Hide, Expand, Contract, and MoveTo. If a Circle method were to call one of those five methods, the compiler would replace the call with the address of one of Circle's own methods.

If no method by a name is defined within an object type, the compiler goes up to the immediate ancestor type, and looks within that type for a method of the name called. If a method by that name is found, the address of the ancestor's method replaces the name in the descendant's method's source code. If no method by that name is found, the compiler continues up to the next ancestor, looking for the named method. If the compiler hits the very first (top) object type, it issues an error message indicating that no such method is defined.

But when a static inherited method is found and used, you must remember that the method called is the method exactly as it was defined and compiled for the ancestor type. If the ancestor's method calls other methods, the methods called will be the ancestor's methods, even if the descendant has methods that override the ancestor's methods.

The methods discussed so far are static methods. They are static for the same reason that static variables are static: The compiler allocates them and resolves all references to them at compile time. As you've seen, objects and static methods can be powerful tools for organizing a program's complexity.

Sometimes, however, they are not the best way to handle methods.

Problems like the one described in the previous section are due to the compile-time resolution of method references. The way out is to be dynamic—and resolve such references at run time. Certain special mechanisms must be in place for this to be possible, but Turbo Pascal provides those mechanisms in its support of virtual methods.

Virtual methods implement an extremely powerful tool for generalization called polymorphism. Polymorphism is Greek for "many shapes," and it is just that: A way of giving an action one name that is shared up and down an object hierarchy, with each
object in the hierarchy implementing the action in a way appropriate to itself.

The simple hierarchy of graphic figures already described provide a good example of polymorphism in action, implemented through virtual methods.

Each object type in our hierarchy represents a different type of figure on the screen: a point or a circle. It certainly makes sense to say that you can show a point on the screen, or show a circle. Later on, if you were to define objects to represent other figures such as lines, squares, arcs, and so on, you could write a method for each that would display that object on the screen. In the new way of object-oriented thinking, you could say that all these graphic figure types had the ability to show themselves on the screen. That much they all have in common.

What is different for each object type is the way it must show itself to the screen. A point is drawn with a point-plotting routine that needs nothing more than an X,Y location and perhaps a color value. A circle needs an entirely separate graphics routine to display itself, taking into account not only X and Y, but a radius as well. Still further, an arc needs a start angle and an end angle, and a more complex drawing algorithm to take them into account.

Any graphic figure can be shown, but the mechanism by which each is shown is specific to each figure. One word, “Show,” is used to show (literally) many shapes.

That’s a good example of what polymorphism is, and virtual methods are how it is done in Turbo Pascal 5.5.

---

Early binding vs. late binding

The difference between a static method call and a virtual method call is the difference between a decision made now and a decision delayed. When you code a static method call, you are in essence telling the compiler, “You know what I want. Go call it.” Making a virtual method call, on the other hand, is like telling the compiler, “You don’t know what I want—yet. When the time comes, ask the instance.”

Think of this metaphor in terms of the MoveTo problem mentioned in the previous section. A call to Circle.MoveTo can only go to one place: the closest implementation of MoveTo up the object hierarchy. In that case, Circle.MoveTo would still call Point's
definition of MoveTo, since Point is the closest up the hierarchy from Circle. Assuming that no descendent type defined its own MoveTo to override Point's MoveTo, any descendent type of Point would still call the same implementation of MoveTo. The decision can be made at compile time and that's all that needs to be done.

When MoveTo calls Show, however, it's a different story. Every figure type has its own implementation of Show, so which implementation of Show is called by MoveTo should depend entirely on what object instance originally called MoveTo. This is why the call to the Show method within the implementation of MoveTo must be a delayed decision: When compiling the code for MoveTo, no decision as to which Show to call can be made. The information isn't available at compile time, so the decision has to be deferred until run time, when the object instance calling MoveTo can be queried.

The process by which static method calls are resolved unambiguously to a single method by the compiler at compile time is early binding. In early binding, the caller and the callee are connected (bound) at the earliest opportunity, that is, at compile time. With late binding, the caller and the callee cannot be bound at compile time, so a mechanism is put into place to bind the two later on, when the call is actually made.

The nature of the mechanism is interesting and subtle, and you'll see how it works a little later.

Object type compatibility

Inheritance somewhat changes Turbo Pascal's type compatibility rules. In addition to everything else, a descendant type inherits type compatibility with all its ancestor types. This extended type compatibility takes three forms:

- between object instances
- between pointers to object instances
- between formal and actual parameters

In all three forms, however, it is critical to remember that type compatibility extends only from descendant to ancestor. In other words, descendant types can be freely used in place of ancestor types, but not vice versa.
Consider these declarations:

```pascal
type
  LocationPtr = ^Location;
  PointPtr = ^Point;
  CirclePtr = ^Circle;
var
  ALocation : Location;
  APoint : Point;
  ACircle : Circle;
  PLocation : LocationPtr;
  PPoint : PointPtr;
  PCircle : CirclePtr;
```

With these declarations, the following assignments are legal:

- `ALocation := APoint;`
- `APoint := ACircle;`
- `ALocation := ACircle;`

The reverse assignments are not legal.

This is a concept new to Pascal, and it might be a little hard to remember, at first, which way the type compatibility goes. Think of it this way: **The source must be able to completely fill the destination.** Descendant types contain everything their ancestor types contain by virtue of inheritance. Therefore a descendant type is either exactly the same size or (usually) larger than its ancestors, but never smaller. Assigning an ancestor object to a descendant object could leave some of the descendant's fields undefined after the assignment, which is dangerous and therefore illegal.

In an assignment statement, only the fields that the two types have in common will be copied from the source to the destination. In the assignment statement

```
ALocation := ACircle;
```

only the `X` and `Y` fields of `ACircle` will be copied to `ALocation`, since `X` and `Y` are all that types `Circle` and `Location` have in common.

Type compatibility also operates between pointers to object types, under the same general rules as with instances of object types: Pointers to descendants can be assigned to pointers to ancestors. Again, given the earlier definitions, these pointer assignments are legal:

- `PPoint := PCircle;`
PLocation := PPoint;
PLocation := PCircle;

Remember, the reverse assignments are not legal.

A formal parameter (either value or var) of a given object type can take as an actual parameter an object of its own, or any descendant type. Given this procedure header,

```pascal
procedure DragIt(Target : Point);
```

actual parameters could legally be of type Point or Circle, but not type Location. Target could also be a var parameter; the same type compatibility rules apply.

However, keep in mind that there's a drastic difference between a value parameter and a var parameter: A var parameter is a pointer to the actual object passed as a parameter, whereas a value parameter is only a copy of the actual parameter. That copy, moreover, only includes the fields and methods included in the formal value parameter's type. This means the actual parameter is literally translated to the type of the formal parameter. A var parameter is more similar to a typecast, in that the actual parameter remains unaltered.

Similarly, if a formal parameter is a pointer to an object type, the actual parameter can be a pointer to that object type or a pointer to any of that object's descendant types. Given this procedure header,

```pascal
procedure Figure.Add(NewFigure : PointPtr);
```

actual parameters could legally be of type PointPtr or CirclePtr, but not type LocationPtr.

-- Polymorphic objects

In reading the previous section, you might have asked yourself: If any descendant type of a parameter's type can be passed in the parameter, how does the user of the parameter know which object type it is receiving? In fact, the user does not know, not directly. The exact type of the actual parameter is unknown at compile time. It could be any one of the object types descended from the var parameter type, and is thus called a polymorphic object.

Now, exactly what are polymorphic objects good for? Primarily, this: Polymorphic objects allow the processing of objects whose type is not known at compile time. This whole notion is so new to the Pascal
way of thinking that an example might not occur to you immediately. (You’ll be surprised, in time, at how natural it begins to seem. That’s when you’ll truly be an object-oriented programmer.)

Suppose you’ve written a graphics drawing toolbox that supports numerous types of figures: points, circles, squares, rectangles, curves, and so on. As part of the toolbox, you want to write a routine that will drag a graphics figure around the screen with the mouse pointer.

The old way would have been to write a separate drag procedure for each type of graphics figure supported by the toolbox. You would have had to write `DragCircle`, `DragSquare`, `DragRectangle`, and so on. Even if the strong typing of Pascal allowed it (and don’t forget, there are always ways to circumvent strong typing), the differences between the types of graphics figures would seem to prevent a truly general dragging routine from being written.

After all, a circle has a radius but no sides, a square has one length of side, a rectangle two different lengths of side, and curves, arrgh....

At this point, clever Turbo Pascal hackers will step forth and say, do it this way: Pass the graphics figure record to procedure `DragIt` as the referent of a generic pointer. Inside `DragIt`, examine a tag field at a fixed offset inside the graphics figure record to determine what sort of figure it is, and then branch using a case statement:

```pascal
    case FigureIDTag of
      Point    : DragPoint;
      Circle   : DragCircle;
      Square   : DragSquare;
      Rectangle: DragRectangle;
      Curve    : DragCurve;
      ...
```

Well, placing seventeen small suitcases inside one enormous suitcase is a slight step forward, but what’s the real problem with this way of doing things?

What if the user of the toolbox defines some new graphics figure type?

What indeed? What if the user designs traffic signs and wants to work with octagons for stop signs? The toolbox does not have an Octagon type, so `DragIt` would not have an Octagon label in its
case statement, and would therefore refuse to drag the new
Octagon figure. If it were presented to DragIt, Octagon would fall
out in the case statement's else clause as an "unrecognized
figure."

Plainly, building a toolbox of routines for sale without source
code suffers from this problem: The toolbox can only work on
data types that it "knows," that is, that are defined by the design-
ers of the toolbox. The user of the toolbox is powerless to extend
the function of the toolbox in directions unanticipated by the
toolbox designers. What the user buys is what the user gets.
Period.

The way out is to use Turbo Pascal's extended type compatibility
rules for objects and design your application to use polymorphic
objects and virtual methods. If a toolbox DragIt procedure is set
up to work with polymorphic objects, it will work with any
objects defined within the toolbox—and any descendant objects
that you define yourself. If the toolbox object types use virtual
methods, the toolbox objects and routines can work with your
custom graphics figures on the figures' own terms. A virtual method
you define today is callable by a toolbox .TPU unit file that was
written and compiled a year ago. Object-oriented programming
makes it possible, and virtual methods are the key.

Understanding how virtual methods make such polymorphic
method calls possible requires a little background on how virtual
methods are declared and used.

**Virtual methods**

A method is made virtual by following its declaration in the
object type with the new reserved word virtual. Remember that if
you declare a method in an ancestor type virtual, all methods of
the same name in any descendant must also be declared virtual to
avoid a compiler error.

Here are the graphics shape objects you've been seeing, properly
virtualized:

```pascal
type
  Location = object
    X, Y : Integer;
  procedure Init(InitX, InitY : Integer);
  function GetX : Integer;
  function GetY : Integer;
end;
```
Every object type that has virtual methods must have a constructor.

We suggest the use of the identifier Init for object constructors.

Warning!  

Point = object(Location)
  Visible : Boolean;
  constructor Init(InitX, InitY : Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  function IsVisible : Boolean;
  procedure MoveTo(NewX, NewY : Integer);
end;

Circle = object(Point)
  Radius : Integer;
  constructor Init(InitX, InitY : Integer;
                   InitRadius : Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure Expand(ExpandBy : Integer); virtual;
  procedure Contract(ContractBy : Integer); virtual;
end;

Notice first of all that the MoveTo method shown in the last iteration of type Circle is gone from Circle's type definition. Circle no longer needs to override Point's MoveTo method with an unmodified copy compiled within its own scope. Instead, MoveTo can now be inherited from Point, with all of MoveTo's nested method calls going to Circle's methods rather than Point's, as happens in an all-static object hierarchy.

Also, notice the new reserved word constructor replacing the reserved word procedure for Point.Init and Circle.Init. A constructor is a special type of procedure that does some of the setup work for the machinery of virtual methods. Furthermore, the constructor must be called before any virtual method is called. Calling a virtual method without previously calling the constructor can cause system lockup, and the compiler has no way to check the order in which methods are called.

Each individual instance of an object must be initialized by a separate constructor call. It is not sufficient to initialize one instance of an object and then assign that instance to additional instances. The additional instances, while they might contain correct data, will not be initialized by the assignment statements, and will lock up the system if their virtual methods are called.

What do constructors construct? Every object type has something called a virtual method table (VMT) in the data segment. The VMT contains the object type's size, and for each of its virtual methods, a pointer to the code implementing that method. What the con-
The default state of \( SR \) is inactive, \( (SR-) \).

Once virtual, always virtual

You'll notice that both \textit{Point} and \textit{Circle} have methods named \texttt{Show} and \texttt{Hide}. All method headers for \texttt{Show} and \texttt{Hide} are tagged as virtual methods with the reserved word \texttt{virtual}. Once an ancestor object type tags a method as \texttt{virtual}, all its descendant types that implement a method of that name must tag that method \texttt{virtual} as well. In other words, a static method can never override a virtual method. If you try, a compiler error will result.

You should also keep in mind that the method heading cannot change in \textit{any} way downward in an object hierarchy once the method is made virtual. You might think of each definition of a virtual method as a gateway to \textit{all} of them. For this reason, the headers for all implementations of the same virtual method must be identical, right down to the number and type of parameters. This is not the case for static methods; a static method overriding another can have different numbers and types of parameters as necessary.

During program development, you might wish to take advantage of a safety net that Turbo Pascal 5.5 places beneath virtual method calls. If the \$R toggle is in its active state, \( ($R+$$) \), all virtual method calls are checked for the initialization status of the instance making the call. If the instance making the call has not been initialized by its constructor, a range check run-time error occurs.

Once you've shaken out a program and are certain that no method calls from uninitialized instances are present, you can speed your code up somewhat by setting the \$R toggle to its inactive state, \( ($R-$$) \). Method calls from uninitialized instances will no longer be checked for, and will probably lock up your system if found.
An example of late binding

To show how to use polymorphic objects with late binding in a Turbo Pascal 5.5 program, let's return to the graphics figures unit described earlier on page 20. The goal is to create a unit that exports several graphics figure objects (like Point and Circle) and a generalized means of dragging any of them around the screen. The unit, named Figures, will be a simple implementation of the graphics toolbox discussed earlier. To demonstrate Figures, let's build a simple program that defines a new figure object type unknown to Figures and then uses virtual methods to drag that new figure type around the screen.

Think about how graphics figures are alike and how they differ. The differences are obvious, and all involve shapes and angles and curves drawn on the screen. In the simple graphics program we'll describe, figures displayed on a screen share these attributes:

- They have a location, given as X,Y. The point within a figure considered to lie at this X,Y position is called the figure's anchor point.
- They can be either visible or invisible, specified by a Boolean value of True (visible) or False (invisible).

If you recall the earlier examples, these are precisely the characteristics of the Location and Point object types. Point, in fact, represents a sort of "grandparent" type from which all graphics figure objects are descended.

The rationale demonstrates an important principle of object-oriented programming: In defining a hierarchy of object types, gather all common attributes into a single type and allow the hierarchy of types to inherit all common elements from that type.

Type Point acts as a template from which its descendant object types can take elements common to all types in the hierarchy. In this example, no object of type Point will ever actually be drawn to the screen, though no harm would come of doing so. (Calling Point.Show would obviously display a point on the screen.) An object type specifically designed to provide inheritable characteristics for its descendants we call an abstract object type. The point of an abstract type is to have descendants, not instances.
Go back to page 35 and read *Point* over once more, this time as a compendium of all the things that graphics figures have in common. *Point* inherits $X$ and $Y$ from the even earlier *Location* type, but *Point* contains $X$ and $Y$ nonetheless, and can bequeath them to its descendant types. Note that none of *Point*’s methods address the shape of a figure, but all figures can be visible or invisible, and be moved around on the screen.

*Point* also has an important function as a “broadcasting station” for changes to the object hierarchy *as a whole*. If some new feature is devised that applies to all graphics figures (color support, for example), it can be added to all object types descended from *Point* simply by adding the new features to *Point*. The new features are instantly callable from any of *Point*’s descendant types. A method for moving a figure to the current position of the mouse pointer, for example, could be added to *Point* without changing any figure-specific methods, since such a method would only affect the two fields $X$ and $Y$.

Obviously, if the new feature must be implemented differently for different figures, there must be a whole family of figure-specific virtual methods added to the hierarchy, each method overriding the one belonging to its immediate ancestor. Color, for example, would require minor changes to *Show* and *Hide* up and down the line, since the syntax of many GRAPH.TPU drawing routines depends on how drawing color is specified.

---

**Procedure or method?**

A major goal in designing the FIGURES.PAS unit is to allow users of the unit to extend the object types defined in the unit—and still make use of all the unit’s features. It is an interesting challenge to create some means of dragging an arbitrary graphics figure around the screen in response to user input.

There are two ways to go about it. The way that might first occur to traditional Pascal programmers is to have FIGURES.PAS export a procedure that takes a polymorphic object as a *var* parameter, and then drags that object around the screen. Such a procedure is shown here:

```pascal
procedure DragIt(var AnyFigure : Point; DragBy : Integer);
var
    DeltaX,DeltaY : Integer;
    FigureX,FigureY : Integer;
```

---

**Chapter 1. All about OOP**
This procedure works fine, but the OOP way of doing it is more elegant (see page 42).

```pascal
begin
  AnyFigure.Show;  { Display figure to be dragged }
  FigureX := AnyFigureGetX;  { Get the initial X,Y of figure }
  FigureY := AnyFigureGetY;

  { This is the drag loop }
  while GetDelta(DeltaX, DeltaY) do
    begin
      { Apply delta to figure X,Y: }
      FigureX := FigureX + (DeltaX * DragBy);
      FigureY := FigureY + (DeltaY * DragBy);
      { And tell the figure to move }
      AnyFigure.MoveTo(FigureX, FigureY);
    end;
end;
```

DragIt calls an additional procedure, GetDelta, that obtains some sort of change in X and Y from the user. It could be from the keyboard, or from a mouse, or a joystick. (For simplicity's sake, our example will obtain input from the arrow keys on the keypad.)

What's important to notice about DragIt is that any object of type Point or any type descended from Point can be passed in the AnyFigure var parameter. Instances of Point or Circle, or any type defined in the future that inherits from Point or Circle, can be passed without complication in AnyFigure.

How does DragIt's code know what object type is actually being passed? It doesn't—and that's OK. DragIt only references identifiers defined in type Point. By inheritance, those identifiers are also defined in any descendant of type Point. The methods GetX, GetY, Show, and MoveTo are just as truly present in type Circle as in type Point, and would be present in any future type defined as a descendant of either.

GetX, GetY, and MoveTo are static methods, which means that DragIt knows the procedure address of each at compile time. Show, on the other hand, is a virtual method. There is a different implementation of Show for both Point and Circle—and DragIt does not know at compile time which implementation is to be called. In brief, when DragIt is called, DragIt looks up the address of the correct implementation of Show in the VMT of the instance passed in AnyFigure. If the instance is a Circle, DragIt calls Circle.Show. If the instance is a Point, DragIt calls Point.Show. The decision as to which implementation of Show will be called is not made until run time, and not, in fact, until the moment in the program when DragIt must call virtual method Show.
Now, *DragIt* works quite well, and if it is exported by the toolbox unit, it can drag any descendant type of *Point* around the screen, whether that type existed when the toolbox was compiled or not. But you have to think a little further: If any object can be dragged around the screen, why not make dragging a feature of the graphics objects themselves?

In other words, why not make *DragIt* a method?

Make it a method!

Indeed. Why pass an object to a procedure to drag the object around the screen? That's old-school thinking. If a procedure can be written to drag any graphics figure object around the screen, then the graphics figure objects ought to be able to drag themselves around the screen.

In other words, procedure *DragIt* really ought to be method *Drag*.

Adding a new method to an existing object hierarchy involves a little thought. How far up the hierarchy should the method be placed? Think about the utility provided by the method and decide how broadly applicable that utility is. Dragging a figure involves changing the location of the figure in response to input from the user. Metaphorically, you might think of a *Drag* method as *MoveTo* with an internal power source. In terms of inheritability, it sits right beside *MoveTo*—any object to which *MoveTo* is appropriate should also inherit *Drag*. *Drag* should thus be added to our abstract object type, *Point*, so that all *Point*'s descendants can share it.

Does *Drag* need to be virtual? The litmus test for making any method virtual is whether the functionality of the method is expected to change somewhere down the hierarchy tree. *Drag* is a closed-ended sort of feature. It only manipulates the *X*, *Y* position of a figure, and one doesn’t imagine that it would become more than that. Therefore, it probably doesn’t need to be virtual.

Use caution in any such decision: If you don’t make *Drag* virtual, you lock out all opportunities for users of FIGURES.PAS to alter it in their efforts to extend FIGURES.PAS. You might not be able to imagine the circumstances under which a user might want to rewrite *Drag*. That doesn’t for a moment mean that such circumstances will not arise.

For example, *Drag* has a joker in it that tips the balance in favor of its being virtual: It deals with *event handling*, that is, the interception of input from devices like the keyboard and mouse, which
occur at unpredictable times yet must be handled when they occur. Event handling is a messy business, and often very hardware-specific. If your user has some input device that does not meld well with Drag as you present it, the user will be helpless to rewrite Drag. Don’t burn any bridges. Make Drag virtual.

The process of converting DragIt to a method and adding the method to Point is almost trivial. Within the Point object definition, Drag is just another method header:

```pascal
Point = object(Location)
  Visible : Boolean;
  constructor Init(InitX, InitY : Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  function IsVisible : Boolean;
  procedure MoveTo(NewX, NewY : Integer);
  procedure Drag(DragBy : Integer); virtual;
end;
```

The position of Drag's method header in the Point object definition is unimportant. Remember, methods can be declared in any order, but data fields must be defined before the first method declaration.

Changing the procedure DragIt to the method Drag is almost entirely a matter of applying Point's scope to DragIt. In the DragIt procedure, you had to specify AnyFigure.Show, AnyFigure.GetX, and so on. Drag is now a part of Point, so you no longer have to qualify method names. AnyFigure.GetX is now simply GetX, and so on. And of course, the AnyFigure var parameter is banished from the parameter line. The implied Self parameter now tells you which object instance is calling Drag.

The complete source code for FIGURES.PAS, including Drag implemented as a virtual method, is shown next:

```pascal
unit Figures;  { Virtual methods & polymorphic objects }
interface
uses Graph, Crt;
type
  Location = object
    X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    function GetX : Integer;
    function GetY : Integer;
```
end;

PointPtr = ^Point;

Point = object(Location)
  Visible : Boolean;
  constructor Init(InitX, InitY : Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  function IsVisible : Boolean;
  procedure MoveTo(NewX, NewY : Integer);
  procedure Drag(DragBy : Integer); virtual;
end;

CirclePtr = ^Circle;

Circle = object(Point)
  Radius : Integer;
  constructor Init(InitX, InitY : Integer;
                  InitRadius : Integer);
      procedure Show; virtual;
      procedure Hide; virtual;
      procedure Expand(ExpandBy : Integer); virtual;
      procedure Contract(ContractBy : Integer); virtual;
end;

implementation

{--------------------------------------------------------I
{ Location’s method implementations: }
{--------------------------------------------------------I

procedure Location.Init(InitX, InitY : Integer);
begin
  X := InitX;
  Y := InitY;
end;

function Location.GetX : Integer;
begin
  GetX := X;
end;

function Location.GetY : Integer;
begin
  GetY := Y;
end;

{--------------------------------------------------------I
{ Point’s method implementations: }
{--------------------------------------------------------I


**constructor** Point.Init(InitX, InitY: Integer);
begin
  Location.Init(InitX, InitY);
  Visible := False;
end;

**destructor** Point.Done;
begin
  Hide;
end;

**procedure** Point.Show;
begin
  Visible := True;
  PutPixel(X, Y, GetColor);
end;

**procedure** Point.Hide;
begin
  Visible := False;
  PutPixel(X, Y, GetBkColor);
end;

**function** Point.IsVisible : Boolean;
begin
  IsVisible := Visible;
end;

**procedure** Point.MoveTo(NewX, NewY: Integer);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;

**function** GetDelta(var DeltaX: Integer;
                       var DeltaY: Integer) : Boolean;
var
  KeyChar: Char;
  Quit: Boolean;
begin
  DeltaX := 0; DeltaY := 0; { 0 means no change in position;  }
  GetDelta := True;    { True means we return a delta    }
  repeat
    KeyChar := ReadKey;  { First, read the keystroke    }
    Quit := True;       { Assume it's a useable key    }
    case Ord(KeyChar) of
      0: begin { 0 means an extended, 2-byte code }
        KeyChar := ReadKey; { Read second byte of code  }
        case Ord(KeyChar) of
          72: DeltaY := -1; { Up arrow; decrement Y  }
  end;
80: DeltaY := 1;  { Down arrow; increment Y }  
75: DeltaX := -1;  { Left arrow; decrement X }  
77: DeltaX := 1;  { Right arrow; increment X }  
else Quit := False;  { Ignore any other code }  
end;  { case }  
end;

13: GetDelta := False;  { CR pressed means no delta }  
else Quit := False;  { Ignore any other keystroke }  
end;  { case }  
until Quit;

procedure Point.Drag(DragBy : Integer);
var
  DeltaX, DeltaY : Integer;
  FigureX, FigureY : Integer;
begin
  Show;
  FigureX := GetX;  { Get the initial position of figure }  
  FigureY := GetY;

  while GetDelta(DeltaX, DeltaY) do
  begin
    FigureX := FigureX + (DeltaX * DragBy);  
    FigureY := FigureY + (DeltaY * DragBy);
    MoveTo(FigureX, FigureY);  { And tell the figure to move }  
  end;
end;

-----------------------------------------------  
{ Circle's method implementations: }  
-----------------------------------------------  
constructor Circle.Init(InitX, InitY : Integer;
  InitRadius : Integer);
begin
  Point.Init(InitX, InitY);
  Radius := InitRadius;
end;
procedure Circle.Show;
begin
  Visible := True;
  Graph.Circle(X, Y, Radius);
end;

procedure Circle.Hide;
var
  TempColor : Word;
begin
  TempColor := Graph.GetColor;
end;
Object extensibility

By now, you should be thinking in terms of building functionality into objects in the form of methods rather than building procedures and passing objects to them as parameters. Ultimately you'll come to design programs in terms of activities that objects can do, rather than as collections of procedure calls that act upon passive data.

It's a whole new world.

The important thing to notice about toolbox units like FIGURES.PAS is that the object types and methods defined in the unit can be distributed to users in linkable .TPU form only, without source code. (Only a listing of the interface portion of the unit need be released.) Using polymorphic objects and virtual methods, the users of the .TPU file can still add features to it to suit their needs.

This novel notion of taking someone else's program code and adding functionality to it without benefit of source code is called extensibility. Extensibility is a natural outgrowth of inheritance: You inherit everything that all your ancestor types have, and then you add what new capability you need. Late binding lets the new meld with the old at run time, so the extension of the existing

```pascal
Graph.SetColor(GetBkColor);
Visible := False;
Graph.Circle(X, Y, Radius);
Graph.SetColor(TempColor);
end;

procedure Circle.Expand(ExpandBy : Integer);
beg
Hide;
Radius := Radius + ExpandBy;
if Radius <0 then Radius := 0;
Show;
end;

procedure Circle.Contract(ContractBy : Integer);
beg
Expand(-ContractBy);
end;

{ No initialization section }
end.
```
code is seamless and costs you no more in performance than a quick trip through the virtual method table.

The following program makes use of the *Figures* unit, and extends it by creating a new graphics figure object, *Arc*, as a descendant type of *Circle*. The object *Arc* could have been written long after FIGURES.PAS was compiled, and yet an object of type *Arc* can make use of inherited methods like *MoveTo* or *Drag* without any special considerations. Late binding and *Arc*’s virtual methods allows the *Drag* method to call *Arc*’s *Show* and *Hide* methods even though those methods might have been written long after *Point.Drag* itself was compiled:

```pascal
program FigureDemo;  { Extending FIGURES.PAS with type Arc }
uses Crt, DOS, Graph, Figures;
type
Arc = object(Circle)
  StartAngle, EndAngle : Integer;
constructor Init(InitX, InitY : Integer;
  InitRadius : Integer;
  InitStartAngle, InitEndAngle : Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
end;
var
GraphDriver : Integer;
GraphMode : Integer;
ErrorCode : Integer;
AnArc : Arc;
ACircle : Circle;
{
  Arc’s method declarations:
}
constructor Arc.Init(InitX,InitY : Integer;
  InitRadius : Integer;
  InitStartAngle, InitEndAngle : Integer);
begin
  Circle.Init(InitX, InitY, InitRadius);
  StartAngle := InitStartAngle;
  EndAngle := InitEndAngle;
end;
procedure Arc.Show;
begin
  Visible := True;
end;
```

---

Chapter 1. All about OOP
Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
end;

procedure Arc.Hide;
var
  TempColor : Word;
begin
  TempColor := Graph.GetColor;
  Graph.SetColor(GetBkColor);
  Visible := False;
  ( Draw the arc in the background color to hide it )
  Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
  SetColor(TempColor);
end;

{--------------------------------------------------------I
{ Main program:                                    )
{--------------------------------------------------------I

begin
  GraphDriver := Detect; { Let the BGI determine what board
                         you're using }
  InitGraph(GraphDriver, GraphMode,'');
  if GraphResult <> GrOK then
    begin
      WriteLn('Halted on graphics error:',
               GraphErrorMsg(GraphDriver));
      Halt();
    end;
  { All descendants of type Point contain virtual methods and
   *must* be initialized before use through a constructor call. }
  ACircle.Init(151, 82, 50);   { Initial X,Y at 151,82 }
  AnArc.Init(151, 82, 25, 0, 90); { Initial radius of 50 pixels }  
                                 { Initial X,Y at 151,82 }
                                 { Start angle: 0; End angle: 90 }
  { Replace AnArc with ACircle to drag a circle instead of an arc. Press Enter to stop dragging and end the program. }
  AnArc.Drag(5);               { Parameter is # of pixels to drag by }
  CloseGraph;
end.

In general, you should make methods virtual. Use static methods only when you want to optimize for speed and memory efficiency. The tradeoff, as you've seen, is in extensibility.
Let's say you are declaring an object named *Ancestor*, and within *Ancestor* you are declaring a method named *Action*. How do you decide whether *Action* should be virtual or static? Here's the rule of thumb: Make *Action* virtual if there is a possibility that some future descendant of *Ancestor* will override *Action*, and you want that future code to be accessible to *Ancestor*.

Now apply this rule to the graphics objects you’ve seen in this chapter. In this case, *Point* is the ancestor object type, and you must decide whether to make its methods static or virtual. Let's consider its *Show*, *Hide*, and *MoveTo* methods. Since each different type of figure has its own means of displaying and erasing itself, *Show* and *Hide* will be overridden by each descendant figure. Moving a graphics figure, however, seems to be the same for all descendants: Call *Hide* to erase the figure, change its X,Y coordinates, and then call *Show* to redisplay the figure in its new location. Since this *MoveTo* algorithm can be applied to any figure with a single anchor point at X,Y, it's reasonable to make *Point.MoveTo* a static method that will be inherited by all descendants of *Point*; but *Show* and *Hide* will be overridden and must be virtual so that *Point.MoveTo* can call its descendants' *Show* and *Hide* methods.

On the other hand, remember that if an object has any virtual methods, a VMT will be created for that object type in the data segment and every object instance will have a link to the VMT. Every call to a virtual method must pass through the VMT, while static methods are called directly. Though the VMT lookup is very efficient, calling a method that is static is still a little faster than calling a virtual one. And if there are no virtual methods in your object, then there is no VMT in the data segment and—more significantly—no link to the VMT in every object instance.

The added speed and memory efficiency of static methods must be balanced against the flexibility that virtual methods allow: extension of existing code long after that code is compiled. Keep in mind that users of your object type might think of ways to use it that you never dreamed of, which is, after all, the whole point.

---

**Dynamic objects**

All the object examples shown so far have had static instances of object types that were named in a var declaration and allocated in the data segment and on the stack.
The use of the word static does not relate in any way to static methods.

Objects can be allocated on the heap and manipulated with pointers, just as the closely related record types have always been in Pascal. Turbo Pascal 5.5 includes some powerful extensions to make dynamic allocation and deallocation of objects easier and more efficient.

Objects can be allocated as pointer referents with the `New` procedure:

```pascal
var
  PCircle : ^Circle;
New(PCircle);
```

As with record types, `New` allocates enough space on the heap to contain an instance of the pointer's base type, and returns the address of that space in the pointer.

If the dynamic object contains virtual methods, it must then be initialized with a constructor call before any calls are made to its methods:

```pascal
PCircle^.Init(600, 100, 30);
```

Method calls can then be made normally, using the pointer name and the reference symbol `^` (a caret) in place of the instance name that would be used in a call to a statically allocated object:

```pascal
OldXPosition := PCircle^.GetX;
```

Turbo Pascal 5.5 extends the syntax of `New` to allow a more compact and convenient means of allocating space for an object on the heap and initializing the object with one operation. `New` can now be invoked with two parameters: the pointer name as the first parameter, and the constructor invocation as the second parameter:

```pascal
New(PCircle, Init(600, 100, 30));
```

When you use this extended syntax for `New`, the constructor `Init` actually performs the dynamic allocation, using special entry code generated as part of a constructor's compilation. The instance name cannot precede `Init`, since at the time `New` is called, the instance being initialized with `Init` does not yet exist. The com-
piler identifies the correct Init method to call through the type of the pointer passed as the first parameter.

New has also been extended to allow it to act as a function returning a pointer value. The parameter passed to New is the type of the pointer to the object rather than the pointer variable itself:

```
type
  ArcPtr = ^Arc;

var
  P弧 : ArcPtr;
  P弧 := New(ArcPtr);
```

Note that with version 5.5, the function-form extension to New applies to all data types, not only to object types:

```
type
  CharPtr = ^Char; { Char is not an object type... }

var
  PChar : CharPtr;
  PChar := New(CharPtr);
```

The function form of New, like the procedure form, can also take the object type's constructor as a second parameter:

```
PArc := New(ArcPtr, Init(600, 100, 25, 0, 90));
```

A parallel extension to Dispose has been defined for Turbo Pascal 5.5, as fully explained in the following sections.

Just like traditional Pascal records, objects allocated on the heap can be deallocated with Dispose when they are no longer needed:

```
Dispose(PCircle);
```

There can be more to getting rid of an unneeded dynamic object than just releasing its heap space, however. An object can contain pointers to dynamic structures or objects that need to be released or "cleaned up" in a particular order, especially when elaborate dynamic data structures are involved. Whatever needs to be done to clean up a dynamic object in an orderly fashion should be gathered together in a single method so that the object can be eliminated with one method call:
We suggest the identifier Done for cleanup methods that "close up shop" once an object is no longer needed.

The Done method should encapsulate all the details of cleaning up its object and all the data structures and objects nested within it.

It is legal and often useful to define multiple cleanup methods for a given object type. Complex objects might need to be cleaned up in different ways depending on how they were allocated or used, or depending on what mode or state the object was in when it was cleaned up.

---

**Destructors**

Turbo Pascal 5.5 provides a special type of method called a destructor for cleaning up and disposing of dynamically allocated objects. A destructor combines the heap deallocation step with whatever other tasks are necessary for a given object type. As with any method, multiple destructors can be defined for a single object type.

A destructor is defined with all the object's other methods in the object type definition:

```pascal
Point = object(Location)
   Visible : Boolean;
   Next : PointPtr;
   constructor Init(InitX, InitY : Integer);
   destructor Done; virtual;
   procedure Show; virtual;
   procedure Hide; virtual;
   function IsVisible : Boolean;
   procedure MoveTo(NewX, NewY : Integer);
   procedure Drag(DragBy : Integer); virtual;
end;
```

Destructors can be inherited, and they can be either static or virtual. Because different shutdown tasks are usually required for different object types, we recommend that destructors *always* be virtual so that in every case the correct destructor will be executed for its object type.

Keep in mind that the reserved word destructor is not needed for every cleanup method, even if the object type definition contains virtual methods. Destructors really operate only on dynamically allocated objects. In cleaning up a dynamically allocated object, the destructor performs a special service: It guarantees that the correct number of bytes of heap memory will always be released. There is, however, no harm in using destructors with statically
allocated objects; in fact, by not giving an object type a destructor, you prevent objects of that type from getting the full benefit of Turbo Pascal's dynamic memory management.

Destructors really come into their own when polymorphic objects must be cleaned up and their heap allocation released. A polymorphic object is an object that has been assigned to an ancestor type by virtue of Turbo Pascal's extended type compatibility rules. In the running example of graphics figures, an instance of object type Circle assigned to a variable of type Point is an example of a polymorphic object. These rules apply to pointers to objects as well; a pointer to Circle can be freely assigned to a pointer to type Point, and the referent of that pointer will also be a polymorphic object.

The term polymorphic is appropriate because the code using the object doesn't know at compile time precisely what type of object is on the end of the string—only that the object will be one of a hierarchy of objects descended from the specified type.

The size of object types differ, obviously. So when it comes time to clean up a polymorphic object allocated on the heap, how does Dispose know how many bytes of heap space to release? No information on the size of the object can be gleaned from a polymorphic object at compile time.

The destructor solves the conundrum by going to the place where the information is stored: in the instance variable's VMT. In every object type's VMT is the size in bytes of the object type. The VMT for any object is available through the invisible Self parameter passed to the method on any method call. A destructor is just a special kind of method, and it receives a copy of Self on the stack when an object calls it. So while an object might be polymorphic at compile time, it is never polymorphic at run time, thanks to late binding.

To perform this late-bound memory deallocation, the destructor must be called as part of the extended syntax for the Dispose procedure:

```pascal
Dispose(PPoint, Done);
```

(Calling a destructor outside of a Dispose call does no automatic deallocation at all.) What happens here is that the destructor of the object pointed to by PPoint is executed as a normal method call. As the last thing it does, however, the destructor looks up the size of its instance type in the instance's VMT, and passes the size
to Dispose. Dispose completes the shutdown by deallocating the correct number of bytes of heap space that had previously belonged to PPoint. The number of bytes released will be correct whether PPoint points to an instance of type Point or to one of Point's descendant types like Circle or Arc.

Note that the destructor method itself can be empty and still perform this service:

```pascal
destructor AnObject.Done;
begin
end;
```

What performs the useful work in this destructor is not the method body but the epilog code generated by the compiler in response to the reserved word destructor. In this, it is similar to a unit that exports nothing, but performs some "invisible" service by executing an initialization section before program startup. The action is all behind the scenes.

The final example program provides some practice in the use of objects allocated on the heap, including the use of destructors for object deallocation. The program shows how a linked list of graphics objects might be created on the heap and cleaned up using destructor calls when no longer required.

Building a linked list of objects requires that each object contain a pointer to the next object in the list. Type Point contains no such pointer. The easy way out would be to add a pointer to Point, and in doing so ensure that all of Point's descendant types also inherit the pointer. However, adding anything to Point requires that you have the source code for Point, and as said earlier, one advantage of object-oriented programming is the ability to extend existing objects without necessarily being able to recompile them.

The solution that requires no changes to Point creates a new object type not descended from Point. Type List is a very simple object whose purpose is to head up a list of Point objects. Because Point contains no pointer to the next object in the list, a simple record type, Node, provides that service. Node is even simpler than List, in that it is not an object, has no methods, and contains no data except a pointer to type Point and a pointer to the next node in the list.
List has a method that allows it to add new figures to its linked list of Node records by inserting a new instance of Node immediately after itself, as a referent to its Nodes pointer field. The Add method takes a pointer to a Point object, rather than a Point object itself. Because of Turbo Pascal 5.5's extended type compatibility, pointers to any type descended from Point can also be passed in the Item parameter to List.Add.

Program ListDemo declares a static variable, AList, of type List, and builds a linked list with three nodes. Each node points to a different graphics figure that is either a Point or one of its descendants. The number of bytes of free heap space is reported before any of the dynamic objects are created, and then again after all have been created. Finally, the whole structure, including the three Node records and the three Point objects, are cleaned up and removed from the heap with a single destructor call to the static List object, AList.

List.Done is well worth a close look. Shutting down a List object involves disposing of three different kinds of structures: the polymorphic graphics figure objects in the list, the Node records that hold the list together, and (if it is allocated on the heap) the List object that heads up the list. The whole process is invoked by a single call to AList's destructor:
The code for the destructor merits examination:

```pascal
destructor List.Done;
var
 N: NodePtr;
begin
 while Nodes <> nil do
 begin
  N := Nodes;
  Dispose(N^.Item, Done);
  Nodes := N^.Next;
  Dispose(N);
 end;
end;
```

The list is cleaned up from the list head by the "hand-over-hand" algorithm, metaphorically similar to pulling in the string of a kite: Two pointers, the Nodes pointer within AList and a working pointer N, alternate their grasp on the list while the first item in the list is disposed of. A dispose call deallocates storage for the first Point object in the list (Item^); then Nodes is advanced to the next Node record in the list by the statement Nodes := N^.Next; the Node record itself is deallocated; and the process repeats until the list is gone.

The important thing to note in the destructor Done is the way the Point objects in the list are deallocated:

```pascal
Dispose(N^.Item, Done);
```

Here, N^.Item is the first Point object in the list, and the Done method called is its destructor. Keep in mind that the actual type of N^.Item^ is not necessarily Point, but could as well be any descendant type of Point. The object being cleaned up is a polymorphic object, and no assumptions can be made about its actual size or exact type at compile time. In the earlier call to Dispose, once Done has executed all the statements it contains, the "invisible" epilog code in Done looks up the size of the object instance being cleaned up in the object's VMT. Done passes that size to Dispose, which then releases the exact amount of heap space the polymorphic object actually occupied.

Remember that polymorphic objects must be cleaned up this way, through a destructor call passed to Dispose, if the correct amount of heap space is to be reliably released.
In the example program, \textit{AList} is declared as a static variable in the data segment. \textit{AList} could as easily have been itself allocated on the heap, and anchored to reality by a pointer of type \textit{ListPtr}. If the head of the list had been a dynamic object too, disposing of the structure would have been done by a destructor call executed within \textit{Dispose}:

\begin{verbatim}
var
  PList : ListPtr;
...
Dispose(PList,Done);
\end{verbatim}

Here, \textit{Dispose} calls the destructor method \textit{Done} to clean up the structure on the heap. Then, once \textit{Done} is finished, \textit{Dispose} deallocates storage for \textit{PList}'s referent, removing the head of the list from the heap as well.

The following program uses the same \texttt{FIGURES.PAS} unit described on page 42. It implements an \textit{Arc} type as a descendant of \textit{Point}, creates a \textit{List} object heading up a linked list of three polymorphic objects compatible with \textit{Point}, and then disposes of the whole dynamic data structure with a single destructor call to \textit{AList.Done}.

\begin{verbatim}
program ListDemo; { Dynamic objects \& destructors }
uses Graph, Figures;
type
  ArcPtr = ^Arc;
  Arc = object(Circle)
    StartAngle, EndAngle : Integer;
  constructor Init(InitX, InitY : Integer;
    InitRadius : Integer;
    InitStartAngle, InitEndAngle : Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
end;

  NodePtr = ^Node;
  Node = record
    Item : PointPtr;
    Next : NodePtr;
  end;

  ListPtr = ^List;
  List = object
    Nodes: NodePtr;
  constructor Init;
  destructor Done; virtual;
\end{verbatim}
procedure Add(Item : PointPtr);
procedure Report;
end;

var
  GraphDriver : Integer;
  GraphMode : Integer;
  Temp : String;
  AList : List;
  PArc : ArcPtr;
  PCircle : CirclePtr;
  RootNode : NodePtr;

{---------------------------------------------------------------------
| Procedures that are not methods:
|---------------------------------------------------------------------}

procedure OutTextLn(TheText : String);
begin
  OutText(TheText);
  MoveTo(0, GetY + 12);
end;

procedure HeapStatus(StatusMessage : String);
begin
  Str(MemAvail : 6, Temp);
  OutTextLn(StatusMessage + Temp);
end;

{---------------------------------------------------------------------
| Arc’s method implementations:
|---------------------------------------------------------------------}

constructor Arc.Init(InitX, InitY : Integer;
  InitRadius : Integer;
  InitStartAngle, InitEndAngle : Integer);
begin
  Circle.Init(InitX, InitY, InitRadius);
  StartAngle := InitStartAngle;
  EndAngle := InitEndAngle;
end;

procedure Arc.Show;
begin
  Visible := True;
  Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
end;

procedure Arc.Hide;
var
  TempColor : Word;
begin
TempColor := Graph.GetColor;
Graph.SetColor(GetBkColor);
Visible := False;
Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
SetColor(TempColor);
end;

{--------------------------------------------------------}
{ List's method implementations: }
{--------------------------------------------------------}

class List
begin
Init

Nodes := nil;
end;

Dest List.Done;
var
N : NodePtr;
begin
while Nodes <> nil do
begin
N := Nodes;
Dispose(N^.Item, Done);
Nodes := N^.Next;
Dispose(N);
end;
end;

Add(Item : PointPtr);
var
N : NodePtr;
begin
New(N);
N^.Item := Item;
N^.Next := Nodes;
Nodes := N;
end;

Report;
var
Current : NodePtr;
begin
Current := Nodes;
while Current <> nil do
begin
Str(Current^.Item^.GetX : 3, Temp);
OutTextLn('X = ' + Temp);
Str(Current^.Item^.GetY : 3, Temp);
OutTextLn('Y = ' + Temp);
Current := Current^.Next;
end;
end;

{--------------------------------------------------------}
{ Main program: }
{--------------------------------------------------------}
begin
{ Let the BGI determine what board you’re using: } InitGraph(GraphDriver, GraphMode,'');
if GraphResult <> GrOK then
  begin
  WriteLn('»Halted on graphics error: ',
    GraphErrorMsg(GraphDriver));
    Halt(1);
  end;
HeapStatus('Heap space before list is allocated: ');
{ Create a list }
AList.Init;
{ Now create and add several figures to it in one operation }
AList.Add(New(ArcPtr, Init(151, 82, 25, 200, 330)));
AList.Add(New(CirclePtr, Init(400, 100, 40)));
AList.Add(New(CirclePtr, Init(305, 136, 5)));
{ Traverse the list and display X,Y of the list’s figures } AList.Report;
HeapStatus('Heap space after list is allocated ');
{ Deallocate the whole list with one destructor call } AList.Done;
HeapStatus('Heap space after list is cleaned up: ');
OutText('Press Enter to end program: ');
ReadLn;
CloseGraph;
end.

Where to now?

As with any aspect of computer programming, you don’t get better at object-oriented programming by reading about it; you get better at it by doing it. Most people, on first exposure to
object-oriented programming, are heard to mutter “I don’t get it” under their breath. The “Aha!” comes later that night when, in the midst of putting their own objects in place, the whole concept comes together in the sort of perfect moment we used to call an epiphany. Like the face of woman emerging from a Rorschach inkblot, what was obscure before at once becomes obvious, and from then on it’s easy.

The best thing to do for your first object-oriented project is to take the FIGURES.PAS unit shown on page 42 (you have it on disk) and extend it. Points, circles, and arcs are by no means enough. Create objects for lines, rectangles, and squares. When you’re feeling more ambitious, create a pie-chart object using a linked list of individual pie-slice figures.

One more subtle challenge is to implement objects with relative position. A relative position is an offset from some base point, expressed as a positive or negative difference. A point at relative coordinates -17,42 is 17 pixels to the left of the base point, and 42 pixels down from that base point. Relative positions are necessary to effectively combine figures into single larger figures, since multiple-figure combination figures cannot always be tied together at each figure’s anchor point. Better to define an RX and RY field in addition to anchor point X,Y, and have the final position of the object on the screen be the sum of its anchor point and relative coordinates.

Once you’ve had your “Aha!,” start building object-oriented concepts into your everyday programming chores. Take some existing utilities you use every day and rethink them in object oriented terms. Take another look at your hodgepodge of procedure libraries and try to see the objects in them—then rewrite the procedures in object form. You’ll find that libraries of objects are much easier to reuse in future projects. Very little of your initial investment in programming effort will ever be wasted. You will rarely have to rewrite an object from scratch. If it will serve as is, use it. If it lacks something, extend it. But if it works well, there’s no reason to throw away any of what’s there.

Conclusion

Object-oriented programming is a direct response to the complexity of modern applications, complexity that has often made many programmers throw up their hands in despair. Inher-
itance and encapsulation are extremely effective means for managing complexity. (It's the difference between having ten thousand insects classified in a taxonomy chart, and ten thousand insects all buzzing around your ears.) Far more than structured programming, object-orientation imposes a rational order on software structures that, like a taxonomy chart, imposes order without imposing limits.

Add to that the promise of the extensibility and reusability of existing code, and the whole thing begins to sound almost too good to be true. Impossible, you think?

Hey, this is Turbo Pascal.

"Impossible" is undefined.
Object-oriented debugging

To meet the needs of the object-oriented revolution, both the Turbo Pascal integrated development environment (IDE) and Turbo Debugger have been enhanced to support object-oriented programming. To use these object-oriented features, you must have version 5.5 of Turbo Pascal and version 1.5 of Turbo Debugger.

Object-oriented debugging in the IDE

Working with objects under the IDE involves two functional areas: stepping and tracing through method calls, and examining object data. The integrated debugger “understands” objects and handles them automatically in a fashion consistent with related language components like procedures and records.

Stepping and tracing method calls

A method call is treated by the IDE as an ordinary procedure or function call. F8 (Step) treats a method call as an indivisible unit, and executes it without displaying the method’s internal code; whereas F7 (Trace) loads the method’s code if it’s available and traces through the method’s statements.

There is no difference between tracing static method calls and tracing virtual method calls. Virtual method calls are resolved at run time, but because debugging happens at run time, there is no
ambiguity, and the integrated debugger always knows the correct method to execute next.

The Call Stack window displays the names of methods prefixed by the object type that defines the method (for example, Circle.Init rather than simply Init).

---

**Objects in the Evaluate window**

When they are displayed in the Evaluate window, objects appear in a fashion very similar to records. All the same format specifiers apply, and all expressions that would be valid for records are valid for objects.

Only the data fields are displayed when the object name as a whole is presented to Evaluate. However, when the specific method name is evaluated, as in

```
ACircle.MoveTo
```

a pointer value is displayed indicating the address of the method’s code. This is true for both static and virtual methods. The integrated debugger handles virtual method lookup transparently through the virtual method table (VMT), and the address of a virtual method for a given object instance is the true address of the correct method code for that instance.

When it is tracing inside a method, the IDE “knows” about the scope and presence of the Self parameter. You can evaluate or watch Self, and you can follow it with format specifiers and field or method qualifiers.

---

**Objects in the Watch window**

An object can be added to the Watch window just as a record can; all expressions that would be valid for records are also valid for objects.

---

**Expressions in the Find Procedure command**

Turbo Pascal 5.5 allows the entry of expressions at the prompt for the Find Procedure command of the Debug menu. To be legal, an expression must evaluate to an address in the code segment. Note that this applies to procedural variables and parameters as well as to object methods.
As with the integrated debugger in Turbo Pascal's integrated development environment (IDE), Turbo Debugger version 1.5 has been enhanced to allow you to debug object-oriented programs. Like the IDE, Turbo Debugger is smart about objects.

During Tracing (F7) or Stepping (F8), Turbo Debugger treats methods just as if they were functions or procedures. F7 traces into the method's source code if it's available, while F8 treats the method call as if it were one statement and steps over it.

Like the IDE, Turbo Debugger correctly handles late binding of virtual methods: It always executes and displays the correct code. And Turbo Debugger's Stack window displays the names of methods prefixed by the object type that defines the method.

The "scope" of a symbol is where the debugger looks for that symbol. Turbo Debugger uses the current cursor position to decide a current scope. (See the section, "Implied scope for expression evaluation," in Chapter 9 of the Turbo Debugger manual.) If no Module window is open, Turbo Debugger derives the current scope from the CS:IP values in the CPU window. If a symbol is not in the current scope, you can fully qualify its "path" and Turbo Debugger will find it for you. The following syntax describes how to fully qualify an identifier's scope. Brackets [ ] indicate optional items, while braces {} indicate optional items that may be repeated:

[Unit.] [ObjectType.Method.] {Proc.} [Var]

Here are some examples that don't involve objects and methods:

- AVar: Variable AVar accessible in the current scope.
- AProc: Procedure AProc accessible in the current scope.
- AProc.AVar: Local variable AVar accessible in procedure AProc accessible in the current scope.
- `AUnit.AProc.AVar`: Local variable `AVar` accessible in procedure `AProc` accessible in unit `AUnit`.
- `AUnit.AProc.ANestedProc.AVar`: Local variable `AVar` accessible in procedure `ANestedProc` accessible in procedure `AProc` accessible in unit `AUnit`.

Here are some examples that involve objects and methods:

- `AnInstance`: Instance `AnInstance` accessible in the current scope.
- `AnInstance.AField`: Field `AField` accessible in instance `AnInstance` accessible in the current scope.
- `AnObjectType.AMethod`: Method `AMethod` accessible in object type `AnObjectType` accessible in the current scope.
- `AnInstance.AMethod`: Method `AMethod` accessible in instance `AnInstance` accessible in the current scope.
- `AUnit.AnInstance.AField`: Field `AMethod` accessible in instance `AnInstance` accessible in unit `AUnit`.
- `AUnit.AnObjectType.AMethod`: Method `AMethod` accessible in object type `AnObjectType` accessible in unit `AUnit`.
- `AUnit.AnObjectType.AMethod.ANestedProc.AVar`: Local variable `AVar` accessible in procedure `ANestedProc` accessible in method `AMethod` accessible in object type `AnObjectType` accessible in unit `AUnit`.

You can enter such qualified identifier expressions anywhere an expression is valid (including in the Evaluate and Watch windows), for example, when you’re changing an expression in an Inspector window or using the local menu in the Module window to Goto a method or procedure address in the source code.

---

**Evaluate Window**

Turbo Debugger’s Evaluate window treats an object instance just like the IDE does: The fields are displayed and any format specifier that can be used in evaluating a record can also be used in evaluating an object instance.

When you’re tracing inside a method, Turbo Debugger knows about the scope and presence of the `Self` parameter. You can evaluate (or watch) `Self`, and you can follow it with format specifiers and field or method qualifiers.
Turbo Debugger also lets you call a method from inside the Evaluate window. Just type the object instance name followed by a dot, followed by the method name, followed by the actual parameters (or empty parentheses if there are no parameters). With these declarations,

```pascal
type
  Point = object
    X, Y : Integer;
    Visible : Boolean;
  constructor Init(InitX, InitY : Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure MoveTo(NewX, NewY : Integer);
end;
```

```pascal
var
  APoint : Point;
```

you could enter any of these expressions in Turbo Debugger’s Evaluate window:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>APoint.X</td>
<td>5 ($5) : Integer</td>
</tr>
<tr>
<td>APoint</td>
<td>(5,23,FALSE) : Point</td>
</tr>
<tr>
<td>APoint.MoveTo</td>
<td>@6F4F:00BE</td>
</tr>
<tr>
<td>APoint.MoveTo(10, 10)</td>
<td>calls method MoveTo</td>
</tr>
<tr>
<td>APoint.Show()</td>
<td>calls method Show</td>
</tr>
</tbody>
</table>

Note that you cannot execute constructor or destructor methods in the Evaluate window.

An object can be added to the Watch window just as a record and the same expressions that can be entered in the Evaluate window can also be entered in the Watch window.

Turbo Debugger provides an entirely new window for examining object hierarchies. You can bring up the Object Hierarchy window by pressing H in the View menu.
The two-paned Object Hierarchy window displays information on object *types* rather than object instances. The left pane lists in alphabetical order the object types used by the module being debugged. The right pane shows all objects in their hierarchies, using a line graphic that places the base object type at the left margin of the pane and displays descendant objects beneath and to the right of the base object, with lines indicating ancestor and descendant relationships.

### The object type list pane

The left pane provides an alphabetical list of all object types used by the current module. It supports an incremental match feature to eliminate the need to cursor through large lists of object types: When the highlight bar is in the left pane, simply start typing the name of the object type you're looking for. At each keypress, the pane will highlight the first object type matching all keys pressed up to that point.

Press *Enter* to open an Object Type Inspector window for the highlighted object type. Object Type Inspectors are described on page 69.

### The local menu

Press *Alt-F10* to display the local menu for the pane. You can use the *Ctrl*-key shortcuts if you've enabled shortcuts with TDINST.

This local menu contains two items:

- **Inspect (*Ctrl*-i):** Displays an object type inspector window for the highlighted object type.
- **Tree (*Ctrl*-T):** Moves to the right pane of the window, in which the object hierarchy tree is displayed, and places the highlight bar on the object type that was highlighted in the left pane.

### The hierarchy tree pane

The right pane displays the hierarchy tree for all objects used by the current module. Ancestor and descendant relationships are indicated by lines, with descendant objects to the right of and below their ancestors.

To locate a single object type in a complex hierarchy tree, go back to the left pane and use the incremental search feature; then choose the Tree item from the local menu to move back into the hierarchy tree. The matched type will be under the highlight bar.

When you press *Enter*, an Object Type Inspector window appears for the highlighted object type.
The hierarchy tree's local menu (Alt-F10 in the right pane) has only one item: Inspect. When you choose it, an Object Type Inspector window appears for the highlighted type. However, a faster and easier method is simply to press Enter when you wish to inspect the highlighted object type.

**The Object Type Inspector window**

Turbo Debugger provides a new type of Inspector window to allow you to inspect the details of an object type: the Object Type Inspector window. The window summarizes type information, but does not reference any particular object instance.

The window is divided into two panes horizontally, with the top pane listing the data fields of the object type, and the bottom pane listing the method names and (if the selected method is a function) the function return type. Use the Tab key to move between the two panes of the Object Type Inspector window.

If the highlighted data field is an object type or a pointer to an object type, pressing Enter opens another Object Type Inspector window for the highlighted type. (This action is identical to selecting the Inspect item in the local menu for this window.) In this way, complex nested structures of objects can be inspected quickly with a minimum of keystrokes.

For brevity's sake, method parameters are not shown in the Object Type Inspector window. To examine method parameters, highlight the method and press Enter. A Method Inspector window will appear. The top pane of the window displays the code address for the object type's implementation of the selected method, and the names and types of all method parameters. The bottom pane of the window indicates whether the method is a procedure or a function.

Pressing Enter from anywhere within the Method Inspector window brings the Module window to the foreground, with the cursor at the code that implements the method being inspected.

As with standard inspectors, Esc closes the current Inspector window and F3 closes them all.

**The local menus**

Pressing Alt-F10 brings up the local menu for either pane. If Ctrl-key shortcuts are enabled (through TDINST), you can get to a local menu item by pressing Ctrl and the first letter of the item. The top pane contains these menu items:
Object Instance Inspector window

Bring up this window by placing your cursor on an object instance in the Module window, then press Ctrl-I.

- Inspect (Ctrl-I): If the highlighted field is an object type or a pointer to one, a new Object Type Inspector window is opened for the highlighted field.

- Hierarchy (Ctrl-H): Opens an Object Hierarchy window for the object type being inspected. The Object Hierarchy window is described on page 67.

- Show Inherited (Ctrl-S): Yes is the default value of this toggle. When it is set to Yes, all data fields and methods are shown, whether they are defined within the type of the inspected object or inherited from an ancestor object type. When it is set to No, only those fields and methods defined within the type of the inspected object are displayed.

These are the local menu items for the bottom pane:

- Inspect (Ctrl-I): A Function Inspector window is opened for the highlighted method. If you press Ctrl-I when the cursor is positioned over the address shown in the Function Inspector window, the Module window is brought to the foreground with the cursor at the code implementing the method being inspected.

- Hierarchy (Ctrl-H): Opens an Object Hierarchy window for the object type being inspected. The Object Hierarchy window is described on page 67.

- Show Inherited (Ctrl-S): Yes is the default value of this toggle. When it is set to Yes, all methods are shown, whether defined within the type of the inspected object or inherited from an ancestor object type. When it is set to No, only those methods defined within the type of the inspected object are displayed.

Object Type Inspector windows provide information about object types, but say nothing about the data contained in a particular object instance at a particular time during program execution. Turbo Debugger provides an extended form of the familiar record inspector window specifically to inspect object instances.

Most Turbo Debugger data record Inspector windows have two panes: a top pane summarizing the record's field names and their current values, and a bottom pane displaying the type of the field highlighted in the top pane. An Object Instance Inspector window provides both of those panes, and also a third pane between them. This new pane summarizes the object instance's methods with the
code address of each. (The code address takes into account polymorphic objects and the VMT.)

Local menus

Each of the top two panes of the Object Instance Inspector window has its own local menu, displayed by pressing Alt-F10 in that pane. Again, you can use the Ctrl-key shortcuts to get to individual menu items if you’ve enabled shortcuts with TDIINST. As with Record Inspector windows, the bottom pane serves only to display the type of the highlighted field, and does not have a local menu.

The top pane, which summarizes the data fields for an object, has the following local menu commands:

- **Range (Ctrl-R):** This command is unchanged from earlier versions. It allows the range of array items to be displayed. If the inspected item is not an array or a pointer, the item cannot be accessed.

- **Change (Ctrl-C):** By choosing this command, you can load a new value into the highlighted data field. This command is also unchanged from earlier versions of Turbo Debugger.

- **Methods (Ctrl-M):** This command is new to Turbo Debugger 1.5. It is a Yes/No toggle, with Yes as the default condition. When it is set to Yes, methods are summarized in the middle pane. When it is set to No, the middle pane does not appear.

- **Show Inherited (Ctrl-S):** Again, an item new to Turbo Debugger 1.5, and also a Yes/No toggle. When it is set to Yes, all data fields and methods are shown, whether they are defined within the type of the inspected object, or inherited from an ancestor object type. When it is set to No, only those fields and methods defined within the type of the inspected object are displayed.

- **Inspect (Ctrl-I):** As with earlier versions of Turbo Debugger, choosing this command opens a Data Inspector window on the highlighted field. Pressing Enter over a highlighted field does the same thing.

- **Descend (Ctrl-D):** This command has not changed from earlier versions of Turbo Debugger. The highlighted item takes the place of the item in the current Inspector window. No new Inspector window is opened. However, you cannot return to the previously inspected field, as you could if you had used the Inspect option.

- **New Expression (Ctrl-N):** No change from earlier versions. This command prompts you for a new data item or expression to
inspect. The new item replaces the current one in the window; it doesn’t open another window.

**Hierarchy (Ctrl-H):** This command is new to Turbo Debugger 1.5. When you choose it, an Object Hierarchy window opens. The full description of this window appears on 67.

The middle pane summarizes the methods of an object. The only difference between the method pane’s local menu and the local menu for the data field (top) pane is the absence of the Change command. Unlike data fields, methods cannot be changed during execution, so there is no need for this command.

The bottom pane is there to display the type of the item highlighted in the upper two windows.

### New error messages

**Constructors and destructors cannot be called.**

You probably tried to evaluate a method that’s either a constructor or a destructor. This is not allowed.

**Not an object Pascal program.**

You tried to open an Object Hierarchy window and there are no objects in your program.
Turbo Pascal 5.5 language definition

The material in this chapter comprises additions to chapters 1 through 11 of the Turbo Pascal Reference Guide for Turbo Pascal 5.0. Use the references in the margin of this chapter to look up related material in your 5.0 manuals.

New reserved words

Turbo Pascal version 5.5 adds the following new reserved words:

- constructor
- destructor
- object
- virtual

User-defined identifiers are not allowed to use the same spelling as these, or the existing reserved words.

Object types

An object type is a structure consisting of a fixed number of components. Each component is either a field, which contains data of a particular type, or a method, which performs an operation on the object. Analogous to a variable declaration, the declaration of a field specifies the data type of the field and an identifier that names the field; and analogous to a procedure or function decla-
ration, the declaration of a method specifies a procedure, function, constructor, or destructor heading.

An object type can inherit components from another object type. If T2 inherits from T1, then T2 is a descendant of T1, and T1 is an ancestor of T2.

Inheritance is transitive, that is, if T3 inherits from T2, and T2 inherits from T1, then T3 also inherits from T1. The domain of an object type consists of itself and all its descendants.

The following code shows examples of object type declarations. These declarations are referred to by other examples throughout this chapter.

```pascal
type
  Point = object
    X, Y : Integer;
  end;
```

Turbo Pascal OOP Guide
Rect = object
  A, B : Point;
  procedure Init(XA, YA, XB, YB : Integer);
  procedure Copy(var R : Rect);
  procedure Move(DX, DY : Integer);
  procedure Grow(DX, DY : Integer);
  procedure Intersect(var R : Rect);
  procedure Union(var R : Rect);
  function Contains(P : Point) : Boolean;
end;

StringPtr = ^String;
FieldPtr = ^Field;

Field = object
  X, Y, Len : Integer;
  Name : StringPtr;
  constructor Copy(var F : Field);
  constructor Init(FX, FY, FLen : Integer; FName : String);
  destructor Done; virtual;
  procedure Display; virtual;
  procedure Edit; virtual;
  function GetStr : String; virtual;
  function PutStr(S : String) : Boolean; virtual;
end;

StrFieldPtr = ^StrField;

StrField = object(Field)
  Value : StringPtr;
  constructor Init(FX, FY, FLen : Integer; FName : String);
  destructor Done; virtual;
  function GetStr : String; virtual;
  function PutStr(S : String) : Boolean; virtual;
  function Get : String;
  procedure Put(S : String);
end;

NumFieldPtr = ^NumField;

NumField = object(Field)
  Value, Min, Max : Longint;
  constructor Init(FX, FY, FLen : Integer; FName : String;
                 FMin, FMax : Longint);
  function GetStr : String; virtual;
  function PutStr(S : String) : Boolean; virtual;
  function Get : Longint;
  procedure Put(N : Longint);
end;

ZipFieldPtr = ^ZipField;

Chapter 3, Turbo Pascal 5.5 language definition 75
ZipField = object(NumField)
  function GetStr : String; virtual;
  function PutStr(S : String) : Boolean; virtual;
end;

Contrary to other types, an object type can be declared only in a type declaration part in the outermost scope of a program or unit. Thus, an object type cannot be declared in a variable declaration part or within a procedure, function, or method block.

The component type of a file type cannot be an object type, or any structured type with an object type component.

The scope of a component identifier extends over the domain of its object type. Furthermore, the scope of a component identifier extends over procedure, function, constructor, and destructor blocks that implement methods of the object type and its descendants. For this reason, the spelling of a component identifier must be unique within an object type and all its descendants and all its methods.

The declaration of a method within an object type corresponds to a forward declaration of that method. Thus, somewhere after the object type declaration, and within the same scope as the object type declaration, the method must be implemented by a defining declaration.

When unique identification of a method is required, a qualified method identifier is used. It consists of an object type identifier, followed by a period (.), followed by a method identifier. Like any other identifier, a qualified method identifier can be prefixed with a unit identifier and a period if required.

Within an object type declaration, a method heading can specify parameters of the object type being declared, even though the declaration is not yet complete. This is illustrated by the Copy, Intersect, and Union methods of the Rect type in the previous example.

Methods are by default static, but can, with the exception of constructor methods, be made virtual through the inclusion of a virtual directive in the method declaration. The compiler resolves calls to static methods at compile time, whereas calls to virtual methods are resolved at run time. The latter is sometimes referred to as late binding.

If an object type declares or inherits any virtual methods, then variables of that type must be initialized through a constructor call.
before any call to a virtual method. Thus, any object type that declares or inherits any virtual methods must also declare or inherit at least one constructor method.

An object type can override (redefine) any of the methods it inherits from its ancestors. If a method declaration in a descendant specifies the same method identifier as a method declaration in an ancestor, then the declaration in the descendant overrides the declaration in the ancestor. The scope of an override method extends over the domain of the descendant in which it is introduced, or until the method identifier is again overridden.

An override of a static method is free to change the method heading in any way it pleases. In contrast, an override of a virtual method must match exactly the order, types, and names of the parameters, and the type of the function result, if any. Furthermore, the override must again include a virtual directive.

An object is instantiated, or created, through the declaration of a variable or typed constant of an object type, or by applying the New standard procedure to a pointer variable of an object type. The resulting object is called an instance of the object type.

```pascal
var
  F : Field;
  Z : ZipField;
  FP : FieldPtr;
  ZP : ZipFieldPtr;
```

Given these variable declarations, \( F \) is an instance of Field, and \( Z \) is an instance of ZipField. Likewise, after applying New to \( FP \) and \( ZP \), \( FP \) will point to an instance of Field, and \( ZP \) will point to an instance of ZipField.

A pointer to an object type is assignment compatible with a pointer to any ancestor object type, therefore during execution of a program, a pointer to an object type might point to an instance of that type, or to an instance of any descendant type.

For example, a pointer of type ZipFieldPtr can be assigned to pointers of type ZipFieldPtr, NumFieldPtr, and FieldPtr, and during execution of a program, a pointer of type FieldPtr might be either nil or point to an instance of Field, StrField, NumField, or ZipField, or any other instance of a descendant of Field.

These pointer assignment compatibility rules also apply to object type variable parameters. For example, the Field.Copy method
might be passed an instance of Field, StrField, NumField, ZipField, or any other instance of a descendant of Field.

A method is activated through a method designator of the form Instance.Method, where Instance is an instance of an object type, and Method is a method of that object type.

For static methods, the declared (compile-time) type of Instance determines which method to activate. For example, the designators F.Init and FP.Init will always activate Field.Init, since the declared type of F and FP is Field.

For virtual methods, the actual (run-time) type of Instance governs the selection. For example, the designator FP>Edit might activate Field/Edit, StrField/Edit, NumField/Edit, or ZipField/Edit, depending on the actual type of the instance pointed to by FP.

In general, there is no way of determining which method will be activated by a virtual method designator. You can develop a routine (such as a forms editor input routine) that activates FP>Edit, and later, without modifying that routine, apply it to an instance of a new, unforeseen descendant type of Field. When extensibility of this sort is desired, you should employ an object type with an open-ended set of descendant types, rather than a record type with a closed set of variants.

Assignment compatibility


The rules of assignment compatibility are extended as follows:

- An object type T2 is assignment compatible with an object type T1 if T2 is in the domain of T1.
- A pointer type P2, pointing to an object type T2, is assignment compatible with a pointer type P1, pointing to an object type T1, if T2 is in the domain of T1.

Object component designators


The format of an object component designator is the same as that of a record field designator; that is, it consists of an instance (a variable reference), followed by a period and a component identifier. A component designator that designates a method is called a method designator. A with statement can be applied to an instance
of an object type, in which case the instance and the period can be omitted in referencing components of the object type.

The instance and the period can also be omitted within any method block, and when they are, the effect is the same as if Self and a period was written before the component reference.

**Dynamic object type variables**

See Chapter 3, "Types," in the Reference Guide. The syntax of the New and Dispose standard procedures has been extended to allow a constructor or destructor call as a second parameter when object type pointers are allocated and disposed. For further details, see the later section “Extensions to New and Dispose” on page 86.

**Instance initialization**

See Chapter 3, "Types," in the Reference Guide. If an object type contains virtual methods, then instances of that object type must be initialized through a constructor call before any call to a virtual method. Here's an example:

```pascal
var
  S : StrField;
begin
  S.Init(1, 1, 25, 'Firstname');
  S.Put('Frank');
  S.Display;
  ...
  S.Done;
end;
```

If S.Init had not been called, then the call to S.Display would cause this example to fail.

The rule of required initialization also applies to instances that are components of structured types. For example,

```pascal
var
  Comment : array[1..5] of StrField;
  I : Integer;
begin
  for I := 1 to 5 do Comment[I].Init(1, I + 10, 40, 'Comment');
  ...
  for I := 1 to 5 do Comment[I].Done;
```

*Chapter 3, Turbo Pascal 5.5 language definition* 79
For dynamic instances, initialization is typically coupled with allocation, and cleanup is typically coupled with deallocation, using the extended syntax of the \textit{New} and \textit{Dispose} standard procedures. Here's an example:

```pascal
var
  SP : StrFieldPtr;
begin
  New(SP, Init(1, 1, 25, 'Firstname'));
  SP^.Put('Frank');
  SP^.Display;
  ...
  Dispose(SP, Done);
end;
```

### Object type constants


The declaration of an object type constant uses the same syntax as the declaration of a record type constant. No value is, or can be, specified for method components. Referring to the earlier object type declarations, here are some examples of object type constants:

```pascal
const
  ZeroPoint : Point = (X : 0; Y : 0);
  ScreenRect : Rect =
    (A : (X : 0; Y : 0); B : (X : 80; Y : 25));
  CountField : NumField = (X : 5; Y : 20; Len : 4; Name : nil;
    Value : 0; Min : -999; Max : 999);
```

Constants of an object type that contains virtual methods need not be initialized through a constructor call—this initialization is handled automatically by the compiler.

### @ with a method


You can apply @ to a qualified method identifier to produce a pointer to the method's entry point.
Function calls


The syntax of a function call has been extended to allow a method designator or a qualified method identifier denoting a function to replace the function identifier.

The discussion of extensions to procedure statements in a later section, “Procedure Statements,” also applies to function calls.

Assignment statements


The rules of object type assignment compatibility allow an instance of an object type to be assigned an instance of any of its descendant types. Such an assignment constitutes a projection of the descendant onto the space spanned by its ancestor. For example, given an instance \( F \) of type \( Field \), and an instance \( Z \) of type \( ZipField \), the assignment \( F := Z \) will copy only the fields \( X, Y, \) \( Len, \) and \( Name \).

Assignment to an instance of an object type does not entail initialization of the instance. Referring to the preceding example, the assignment \( F := Z \) does not mean that a constructor call for \( F \) can be omitted.

Procedure statements


The syntax of a procedure statement has been extended to allow a method designator denoting a procedure, constructor, or destructor to replace the procedure identifier.

The instance denoted by the method designator serves two purposes. First, in the case of a virtual method, the actual (run-time) type of the instance determines which implementation of the method is activated. Second, the instance itself becomes an implicit actual parameter of the method; it corresponds to a formal variable parameter named \( Self \) that possesses the type corresponding to the activated method.

Within a method, a procedure statement allows a qualified method identifier to denote activation of a specific method. The object type given in the qualified identifier must be the same as
the method's object type, or an ancestor of it. This type of activation is called a qualified activation.

The implicit Self parameter of a qualified activation becomes the Self of the method containing the call. A qualified activation never employs the virtual method dispatch mechanism—the call is always static, and always invokes the specified method.

A qualified activation is generally used within an override method to activate the overridden method. Referring to the types declared earlier, here are some examples of qualified activations:

```pascal
constructor NumField.Init(FX, FY, FLen : Integer;
  FName : String; FMin, FMax : Longint);
begin
  Field.Init(FX, FY, FLen, FName);
  Value := 0;
  Min := FMin;
  Max := FMax;
end;

function ZipField.PutStr(S : String) : Boolean;
begin
  PutStr := (Length(S) = 5) and NumField.PutStr(S);
end;
```

As these examples demonstrate, a qualified activation allows an override method to "reuse" the code of the method it overrides.

Case statements

See Chapter 7, "Statements," in the Reference Guide. The case statement previously did not allow the selector to be of type Word. This restriction is now gone, and a case selector may be of any byte-sized or word-sized ordinal type.

With statements

See Chapter 7, "Statements," in the Reference Guide. The with statement has been extended to accept object types as well as record types.
Method declarations

The declaration of a method within an object type corresponds to a forward declaration of that method. Thus, somewhere after the object type declaration, and within the same scope as the object type declaration, the method must be implemented by a defining declaration.

For procedure and function methods, the defining declaration takes the form of a normal procedure or function declaration, with the exception that the procedure or function identifier in this case is a qualified method identifier.

For constructor methods and destructor methods, the defining declaration takes the form of a procedure method declaration, except that the procedure keyword is replaced by a constructor or destructor keyword.

A method's defining declaration can optionally repeat the formal parameter list of the method heading in the object type. The defining declaration's method heading must in that case match exactly the order, types, and names of the parameters, and the type of the function result, if any.

In the defining declaration of a method, there is always an implicit parameter with the identifier Self, corresponding to a formal variable parameter that possesses the object type. Within the method block, Self represents the instance whose method component was designated to activate the method. Thus, any changes made to the values of the fields of Self are reflected in the instance.

The scope of a component identifier in an object type extends over any procedure, function, constructor, or destructor block that implements a method of the object type. The effect is the same as if the entire method block was embedded in a with statement of the form

```pascal
with Self do begin ... end
```

For this reason, the spellings of component identifiers, formal method parameters, Self, and any identifiers introduced in a method implementation must be unique.

Here are some examples of method implementations:

```pascal
procedure Rect.Intersect(var R : Rect);
begin
```
if A.X < R.A.X then A.X := R.A.X;
if A.Y < R.A.Y then A.Y := R.A.Y;
if B.X > R.B.X then B.X := R.B.X;
if B.Y > R.B.Y then B.Y := R.B.Y;
if (A.X >= B.X) or (A.Y >= B.Y) then Init(0, 0, 0, 0);
end;

procedure Field.Display;
begin
  GotoXY(X, Y);
  Write(Name^, ', ', GetStr);
end;

function NumField.PutStr(S : String) : Boolean;
var
  E : Integer;
begin
  Val(S, Value, E);
  PutStr := (E = 0) and (Value >= Min) and (Value <= Max);
end;

Constructors and destructors

Constructors and destructors are specialized forms of methods. Used in connection with the extended syntax of the New and Dispose standard procedures, constructors and destructors have the ability to allocate and deallocate dynamic objects. In addition, constructors have the ability to perform the required initialization of objects that contain virtual methods. Like other methods, constructors and destructors can be inherited, and an object can have any number of constructors and destructors.

Constructors are used to initialize newly instantiated objects. Typically, the initialization is based on values passed as parameters to the constructor. Constructors cannot be virtual, because the virtual method dispatch mechanism depends on a constructor first having initialized the object.

Here are some examples of constructors:

constructor Field.Copy(var F : Field);
begin
  Self := F;
end;

constructor Field.Init(FX, FY, FLen : Integer; FName : String);
begin
  X := FX;
end;
Destructors can be virtual, and often are. Destructors seldom take any parameters.

\begin{verbatim}
Y := FY;
Len := FLen;
GetMem(Name, Length(FName) + 1);
Name^ := FName;
end;

constructor StrField.Init(FX, FY, FLen : Integer; FName : String);
begin
  Field.Init(FX, FY, FLen, FName);
  GetMem(Value, Len);
  Value^ := '';
end;
\end{verbatim}

The first action of a constructor of a descendant type, such as the preceding `StrField.Init`, is almost always to call its immediate ancestor's corresponding constructor to initialize the inherited fields of the object. Having done that, the constructor then initializes the fields of the object that were introduced in the descendant.

Destructors are the counterparts of constructors, and are used to clean up objects after their use. Typically, the cleanup consists of disposing any pointer fields in the object.

Here are some examples of destructors:

\begin{verbatim}
destructor Field.Done;
begin
  FreeMem(Name, Length(Name^) + 1);
end;
destructor StrField.Done;
begin
  FreeMem(Value, Len);
  Field.Done;
end;
\end{verbatim}

A destructor of a descendant type, such as the preceding `StrField.Done`, typically first disposes the pointer fields introduced in the descendant, and then, as its last action, calls the corresponding destructor of its immediate ancestor to dispose any inherited pointer fields of the object.

### Variable parameters


The rules of object type assignment compatibility also apply to object type variable parameters: For a formal parameter of type `T1`, the actual parameter might be of type `T2` if `T2` is in the domain of `T1`. For example, the `Field.Copy` method might be passed an in-
stance of Field, StrField, NumField, ZipField, or any other instance of a descendant of Field.

Extensions to New and Dispose


The New and Dispose standard procedures have been extended to allow a constructor call or destructor call as a second parameter for allocating or disposing a dynamic object type variable. The syntax is

\[
\text{New}(P, \text{Construct})
\]

and

\[
\text{Dispose}(P, \text{Destruct})
\]

where \(P\) is a pointer variable, pointing to an object type, and Construct and Destruct are calls to constructors and destructors of that object type. For New, the effect of the extended syntax is the same as executing

\[
\text{New}(P); \\
P^.\text{Construct};
\]

And for Dispose, the effect of the extended syntax is the same as executing

\[
P^.\text{Destruct}; \\
\text{Dispose}(P);
\]

Without the extended syntax, occurrences of such "pairs" of a call to New followed by a constructor call, and a destructor call followed by a call to Dispose would be very common. The extended syntax improves readability, and also generates shorter and more efficient code.

The following illustrates the use of the extended New and Dispose syntax:

\[
\begin{align*}
\text{var} \\
\text{SP : StrFieldPtr;} \\
\text{ZP : ZipFieldPtr;} \\
\text{begin} \\
\text{New}(\text{SP}, \text{Init}(1, 1, 25, 'Firstname')); \\
\text{New}(\text{ZP}, \text{Init}(1, 2, 5, 'Zip code', 0, 99999)); \\
\text{SP^.Edit;} \\
\text{ZP^.Edit;} \\
\end{align*}
\]

...
An additional extension allows `New` to be used as a \textit{function}, which allocates and returns a dynamic variable of a specified type. The syntax is

\texttt{New(T)}

or

\texttt{New(T, Construct)}

In the first form, \( T \) can be any pointer type. In the second form, \( T \) must point to an object type, and \textit{Construct} must be a call to a constructor of that object type. In both cases the type of the function result is \( T \).

Here's an example:

\begin{verbatim}
var
  F1, F2 : FieldPtr;
begin
  F1 := New(StrFieldPtr, Init(1, 1, 25, 'Firstname'));
  F2 := New(ZipFieldPtr, Init(1, 2, 5, 'Zip code', 0, 99999));
  ...
  WriteLn(F1^.GetStr);   { calls StrField.GetStr }
  WriteLn(F2^.GetStr);   { calls ZipField.GetStr }
  ...
  Dispose(F2, Done);     { calls Field.Done }
  Dispose(F1, Done);     { calls StrField.Done }
end;
\end{verbatim}

Notice that even though \( F1 \) and \( F2 \) are of type \textit{FieldPtr}, the extended pointer assignment compatibility rules allow \( F1 \) and \( F2 \) to be assigned a pointer to any descendant of \textit{Field}; and since \textit{GetStr} and \textit{Done} are virtual methods, the virtual method dispatch mechanism will correctly call \textit{StrField.GetStr}, \textit{ZipField.GetStr}, \textit{Field.Done}, and \textit{StrField.Done}, respectively.

\textbf{Compiler directive conditional symbols}

The \texttt{VER50} conditional symbol, which is automatically defined by Turbo Pascal 5.0, has been replaced by \texttt{VER55} in Turbo Pascal 5.5.
This chapter describes the new features found in Turbo Pascal 5.5's overlay manager. The new Overlay unit is fully compatible with Turbo Pascal 5.0's Overlay unit, so any existing overlaid applications can simply be recompiled.

Overlay buffer management

The Turbo Pascal 5.0 overlay buffer is best described as a ring buffer that has a head pointer and a tail pointer. Overlays are always loaded at the head of the buffer, pushing "older" ones toward the tail. When the buffer becomes full (that is, when there is not enough free space between the head and the tail), overlays are disposed at the tail to make room for new ones.

Since ordinary memory is not circular in nature, the actual implementation of the overlay buffer involves a few more steps in order to make the buffer appear to be a ring. Figure 4 illustrates the process. The figure shows a progression of overlays being loaded into an initially empty overlay buffer. Overlay A is loaded first, followed by B, then C, and finally D. Shaded areas indicate free buffer space.
As you can see, a couple of interesting things happen in the transition from step 3 to step 4. First, the head pointer wraps around to the bottom of the overlay buffer, causing the overlay manager to slide all loaded overlays (and the tail pointer) upward. This sliding is required to always keep the free area located between the head pointer and the tail pointer. Second, in order to load overlay D, the overlay manager has to dispose overlay A from the tail of the buffer. Overlay A in this case is the least recently loaded overlay, and therefore the best choice for disposal when something has to go. The overlay manager continues to dispose overlays at the tail to make room for new ones at the head, and each time the head pointer wraps around, the sliding operation is repeated.

This is how Turbo Pascal 5.0's overlay manager operates, and is also the default mode of operation for Turbo Pascal 5.5's overlay manager. New in Turbo Pascal 5.5, however, is an optional optimization of the overlay management algorithm.

Imagine that overlay A contains a number of frequently used routines. Even though these routines are used all the time, A will still occasionally be thrown out of the overlay buffer, only to be reloaded again shortly afterward. The problem here is that the overlay manager knows nothing about the frequency of calls to
routines in A—all it knows is that when a call is made to a routine in A and A is not in memory, it has to load A. One solution to this problem might be to trap every call to routines in A, and then at each call move A to the head of the overlay buffer to reflect its new status as the most recently used overlay. Such call interception is unfortunately very costly in terms of execution speed, and may in some cases slow down the application even more than the additional overlay load operations.

Turbo Pascal 5.5 provides a compromise solution that incurs practically no performance overhead and still maintains a high degree of success in identifying frequently used overlays that shouldn’t be unloaded: When an overlay gets close to the tail of the overlay buffer, it is put on “probation.” If, during this probationary period, a call is made to a routine in the overlay, it is “reprieved,” and will not be disposed when it reaches the tail of the overlay buffer. Instead, it is simply moved to the head of the buffer, and thus gets another free ride around the overlay buffer ring. If, on the other hand, no calls are made to an overlay during its probationary period, indicating less frequent use, the overlay is disposed of when it reaches the tail of the overlay buffer.

The net effect of the probation/reprieval scheme is that frequently used overlays are kept in the overlay buffer, at the cost of intercepting just one call every time the overlay gets close to the tail of the overlay buffer.

Two new overlay manager routines, OvrSetRetry and OvrGetRetry, control the probation/reprieval mechanism. OvrSetRetry sets the size of the area in the overlay buffer to keep on probation, and OvrGetRetry returns the current setting. If an overlay falls within the last OvrGetRetry bytes before the overlay buffer tail, it is automatically put on probation. Any free space in the overlay buffer is considered part of the probation area.

### Variables

This section describes the new variables that Turbo Pascal 5.5 adds to the Overlay unit.

**OvrTrapCount**

```pascal
var OvrTrapCount : Word;
```
Each time a call to an overlaid routine is intercepted by the overlay manager, either because the overlay is not in memory or because the overlay is on probation, the `OvrTrapCount` variable is incremented. The initial value of `OvrTrapCount` is 0.

### OvrLoadCount

```pascal
var OvrLoadCount : Word;
```

Each time an overlay is loaded, the `OvrLoadCount` variable is incremented. The initial value of `OvrLoadCount` is zero.

By examining `OvrTrapCount` and `OvrLoadCount` (for example, in the debugger’s watch window) over identical runs of an application, you can monitor the effect of different probation area sizes (set with `OvrSetRetry`) to find the optimal size for your particular application.

### OvrFileMode

```pascal
var OvrFileMode : Byte;
```

The `OvrFileMode` variable determines the access code to pass to DOS when the overlay file is opened. The default `OvrFileMode` is 0, corresponding to read-only access. By assigning a new value to `OvrFileMode` before calling `OvrInit`, you can change the access code, for example, to allow shared access on a network system. For further details on access code values, refer to your DOS Programmer’s Reference Manual.

### OvrReadBuf

```pascal
type OvrReadFunc = function(OvrSeg : Word) : Integer;
var OvrReadBuf : OvrReadFunc;
```

The `OvrReadBuf` procedure variable allows you to intercept overlay load operations, for example, to implement error handling or to check that a removable disk is present. Whenever the overlay manager needs to read an overlay, it calls the function whose address is stored in `OvrReadBuf`. If the function returns zero, the overlay manager assumes that the operation was successful; if the function result is nonzero, run-time error 209 is generated. The `OvrSeg` parameter indicates what overlay to load, but as you’ll see later, you never need to access this information.
To install your own overlay read function, you must first save the previous value of $OvrReadBuf$ in a variable of type $OvrReadFunc$, and then assign your overlay read function to $OvrReadBuf$. Within your read function, you should call the saved read function to perform the actual load operation. Any validations you want to perform, such as checking that a removable disk is present, should go before the call to the saved read function, and any error checking should go after the call.

The code to install an overlay read function should go right after the call to $OvrInit$, at which point $OvrReadBuf$ will contain the address of the default disk read function.

If you also call $OvrInitEMS$, it uses your read function to read overlays from disk into EMS memory, and if no errors occur, it stores the address of the default EMS read function in $OvrReadBuf$. If you also wish to override the EMS read function, simply repeat the installation process after the call to $OvrInitEMS$.

The default disk read function returns zero in case of success, or a DOS error code in case of failure. Likewise, the default EMS read function returns 0 in case of success, or an EMS error code (ranging from $80$ through $FF$) in case of failure. For details on DOS error codes, refer to the "Run-time Errors" section in Appendix D of the Turbo Pascal Reference Guide. For details on EMS error codes, refer to the Lotus/Intel/Microsoft Expanded Memory Specification.

The following code fragment demonstrates how to write and install an overlay read function. The new overlay read function repeatedly calls the saved overlay read function until no errors occur. Any errors are passed to the $DOSError$ or $EMSError$ procedures (not shown here) so that they can present the error to the user. Notice how the $OvrSeg$ parameter is just passed on to the saved overlay read function, and never directly handled by the new overlay read function.

```pascal
uses Overlay;
var
    SaveOvrRead : OvrReadFunc;
    UsingEMS : Boolean;

{$F+}$

function MyOvrRead(OvrSeg : Word) : Integer;
var
    E : Integer;
```
begin
    repeat
        E := SaveOvrRead(OvrSeg);
        if E <> 0 then
            if UsingEMS then
                EMSError(E) else DOSError(E);
        until E = 0;
        MyOvrRead := 0;
    end;

{$F-}
begin
    OvrInit('MYPROG.OVR');
    SaveOvrRead := OvrReadBuf;  { Save disk default }
    OvrReadBuf := MyOvrRead;    { Install ours }
    UsingEMS := False;
    OvrInitEMS;
    SaveOvrRead := OvrReadBuf;  { Save EMS default }
    OvrReadBuf := MyOvrRead;    { Install ours }
    UsingEMS := True;
    ...
end.

Procedures and functions

This section describes the new procedures and functions that Turbo Pascal 5.5 adds to the Overlay unit.

OvrSetRetry

procedure OvrSetRetry(Size : Longint);

The OvrSetRetry procedure sets the size of the “probation area” in the overlay buffer. If an overlay falls within the Size bytes before the overlay buffer tail, it is automatically put on probation. Any free space in the overlay buffer is considered part of the probation area. For reasons of compatibility with earlier versions of the overlay manager, the default probation area size is zero, which effectively disables the probation/reprieval mechanism. Here’s an example of how to use OvrSetRetry:

OvrInit('MYPROG.OVR');
OvrSetBuf(BufferSize);
OvrSetRetry(BufferSize div 3);
There is no empirical formula for determining the optimal size of the probationary area—however, experiments have shown that values ranging from one-third to one-half of the overlay buffer size provide the best results.

**OvrGetRetry**

```pascal
function OvrGetRetry : Longint;
```

The *OvrGetRetry* function returns the current size of the probation area, that is, the value last set with *OvrSetRetry*.

### Overlays in .EXE files

Turbo Pascal 5.5 allows you to store your overlays at the end of your application's .EXE file rather than in a separate .OVR file. To attach an .OVR file to the end of an .EXE file, use the DOS COPY command with a /B command line switch, for example,

```
COPY/B MYPROG.EXE + MYPROG.OVR
```

You must make sure that the .EXE file was compiled *without* Turbo Debugger debug information. Thus in the IDE, make sure that Debug/Standalone Debugging is set to Off; with the command-line version of the compiler, make sure not to specify a /V switch.

To read overlays from the end of an .EXE file instead of from a separate .OVR file, simply specify the .EXE file name in the call to *OvrInit*. If you are running under DOS 3.x, you can use the **ParamStr** standard function to obtain the name of the .EXE file, for example,

```pascal
OvrInit(ParamStr(0));
```
This chapter is an addendum to Chapter 15, “Inside Turbo Pascal,” in the Turbo Pascal 5.0 Reference Guide.

Internal data format of objects

The internal data format of an object resembles that of a record. The fields of an object are stored in order of declaration, as a contiguous sequence of variables. Any fields inherited from an ancestor type are stored before the new fields defined in the descendant type.

If an object type defines virtual methods, constructors, or destructors, the compiler allocates an extra field in the object type. This 16-bit field, called the virtual method table (VMT) field, is used to store the offset of the object type’s VMT in the data segment. The VMT field immediately follows after the ordinary fields in the object type. When an object type inherits virtual methods, constructors, or destructors, it also inherits a VMT field, so an additional one is not allocated.

Initialization of the VMT field of an instance is handled by the object type’s constructor(s). A program never explicitly initializes or accesses the VMT field.

The following examples illustrate the internal data formats of object types.
Figure 5.1 shows layouts of instances of Location, Point, and Circle; each box corresponds to one word of storage.

Because Point is the first type in the hierarchy that introduces virtual methods, the VMT field is allocated right after the Color field.
stored in the initialized part of the program’s data segment. There is only one VMT per object type (not one per instance), but two distinct object types never share a VMT, no matter how identical they appear to be. VMTs are built automatically by the compiler, and are never directly manipulated by a program. Likewise, pointers to VMTs are automatically stored in object type instances by the object type’s constructor(s) and are never directly manipulated by a program.

The first word of a VMT contains the size of instances of the associated object type; this information is used by constructors and destructors to determine how many bytes to allocate or dispose of, using the extended syntax of the New and Dispose standard procedures.

The second word of a VMT contains the negative size of instances of the associated object type; this information is used by the virtual method call validation mechanism to detect uninitialized objects (instances for which no constructor call has been made), and to check the consistency of the VMT. When virtual call validation is enabled (using the \texttt{[$R+]}) compiler directive, which has been expanded to include virtual method checking), the compiler generates a call to a VMT validation routine before each virtual call. The VMT validation routine checks that the first word of the VMT is not zero, and that the sum of the first and the second word is zero. If either check fails, run-time error 210 is generated.

Enabling range-checking and virtual method call checking slows down your program, and makes it somewhat larger, so use the \texttt{[$R+] state only when debugging, and switch to the \texttt{[$R-]} state for the final version of the program.

Finally, starting at offset 4 in the VMT, comes a list of 32-bit method pointers, one per virtual method in the object type, in order of declaration. Each slot contains the address of the corresponding virtual method’s entry point.

Figure 5.2 shows the layouts of the VMTs of the \textit{Point} and \textit{Circle} types (the \textit{Location} type has no VMT, since it contains no virtual methods, constructors, or destructors); each small box corresponds to one word of storage, and each large box corresponds to two words of storage.
Notice how `Circle` inherits the `Done` and `MoveTo` methods from `Point`, and how it overrides the `Show` and `Hide` methods.

As mentioned already, an object type's constructors contain special code that stores the offset of the object type's VMT in the instance being initialized. For example, given an instance `P` of type `Point`, and an instance `C` of type `Circle`, a call to `P.Init` will automatically store the offset of `Point`'s VMT in `P`'s VMT field, and a call to `C.Init` will likewise store the offset of `Circle`'s VMT in `C`'s VMT field. This automatic initialization is part of a constructor's entry code, so when control arrives at the `begin` of the constructor's statement part, the VMT field `Self` will already have been set up. Thus, if the need arises, a constructor can make calls to virtual methods.

### The SizeOf standard function

When applied to an instance of an object type that has a VMT, `SizeOf` returns the size stored in the VMT. Thus, for object types that have a VMT, `SizeOf` always returns the actual size of the instance, rather than the declared size.
Turbo Pascal 5.5 adds a new standard function, TypeOf, which returns a pointer to an object type’s VMT. TypeOf takes a single parameter, which can be either an object type identifier or an object type instance. In both cases, the result, of type Pointer, is a pointer to the object type’s VMT. TypeOf can be applied only to object types that have a VMT—all other types result in an error.

The TypeOf function can be used to test the actual type of an instance. For example,

```pascal
if TypeOf(Self) = TypeOf(Point) then ...
```

### Virtual method calls

To call a virtual method, the compiler generates code that picks up the VMT address from the VMT field in the object, and then calls via the slot associated with the method. For example, given a variable PP of type PointPtr, the call PP^Show generates the following code:

```pascal
les di, PP ;Load PP into ES:DI
push es  ;Pass as Self parameter
push di
mov di, es:[di+6] ;Pick up VMT offset from VMT field
call DWORD PTR [di+8] ;Call VMT entry for Show
```

The type compatibility rules of object types allow PP to point at a Point or a Circle, or at any other descendant of Point. And if you examine the VMTs shown here, you’ll see that for a Point, the entry at offset 8 in the VMT points to Point.Show, whereas for a Circle, it points to Circle.Show. Thus, depending upon the actual run-time type of PP, the CALL instruction calls Point.Show or Circle.Show, or the Show method of any other descendant of Point.

If Show had been a static method, this code would have been generated for the call to PP^Show:

```pascal
les di, PP ;Load PP into ES:DI
push es  ;Pass as Self parameter
push di
call Point.Show ;Directly call Point.Show
```

Here, no matter what PP points to, the code will always call the Point.Show method.
Method calling conventions

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter, called **Self**, that corresponds to a **var** parameter of the same type as the method’s object type. The **Self** parameter is always passed as the last parameter, and always takes the form of a 32-bit pointer to the instance through which the method is called. For example, given a variable **PP** of type **PointPtr** as defined earlier, the call **PP^MoveTo(10, 20)** is coded as follows:

```
mov ax,10          ;Load 10 into AX
push ax           ;Pass as PX parameter
mov ax,20          ;Load 20 into AX
push ax           ;Pass as PY parameter
les di,PP          ;Load PP into ES:DI
push es           ;Pass as Self parameter
push di
mov di,es:[di+6]   ;Pick up VMT offset from VMT field
call DWORD PTR [di+16] ;Call VMT entry for MoveTo
```

Upon returning, a method must remove the **Self** parameter from the stack, just as it must remove any normal parameters.

Methods always use the FAR CALL model, regardless of the setting of the $F$ compiler directive.

Constructors and destructors

Constructors and destructors use the same calling conventions as normal methods, except that an additional word-sized parameter, called the **VMT** parameter, is passed on the stack just before the **Self** parameter.

For constructors, the VMT parameter contains the VMT offset to store in **Self**'s VMT field in order to initialize **Self**.

Furthermore, when a constructor is called to allocate a dynamic object, using the extended syntax of the **New** standard procedure, a nil pointer is passed in the **Self** parameter. This causes the constructor to allocate a new dynamic object, the address of which is passed back to the caller in DX:AX when the constructor returns. If the constructor could not allocate the object, a nil pointer is returned in DX:AX. (See “Constructor error recovery” on page 106.)
Finally, when a constructor is called using a qualified method identifier (that is, an object type identifier), followed by a period and a method identifier, a value of zero is passed in the VMT parameter. This indicates to the constructor that it should not initialize the VMT field of Self.

For destructors, a 0 in the VMT parameter indicates a normal call, and a nonzero value indicates that the destructor was called using the extended syntax of the Dispose standard procedure. This causes the destructor to deallocate Self just before returning (the size of Self is found by looking at the first word of Self's VMT).

Assembly language methods

Method implementations written in assembly language can be linked with Turbo Pascal programs using the $L$ compiler directive and the external keyword. The declaration of an external method in an object type is no different than that of a normal method; however, the implementation of the method lists only the method header followed by the reserved word external.

In an assembly language source text, an @ is used instead of a period (.) to write qualified identifiers (the period already has a different meaning in assembly language, and cannot be part of an identifier). For example, the Pascal identifier Rect.Init is written as Rect@Init in assembly language. The @ syntax can be used to declare both PUBLIC and EXTRN identifiers.

As an example of assembly language methods, we've implemented a simple Rect object.

```pascal
type
  Rect = object
    X1, Y1, X2, Y2: Integer;
    procedure Init(XA, YA, XB, YB: Integer);
    procedure Union(var R: Rect);
    function Contains(X, Y: Integer): Boolean;
  end;
```

A Rect represents a rectangle bounded by four coordinates, X1, Y1, X2, and Y2. The upper left corner of a rectangle is defined by X1 and Y1, and the lower right corner is defined by X2 and Y2. The Init method assigns values to the rectangle's coordinates; the Union method calculates the smallest rectangle that contains both the rectangle itself and another rectangle; and the Contains
method returns *True* if a given point is within the rectangle, or *False* if not. Other methods, such as moving, resizing, calculating intersections, and testing for equality, could easily be implemented to make *Rect* a more useful object.

The Pascal implementations of *Rect*'s methods list only the method header followed by an *external* keyword.

```
{$L RECT}

procedure Rect.Init(XA, YA, XB, YB: Integer); external;
procedure Rect.Union(var R: Rect); external;
function Rect.Contains(X, Y: Integer): Boolean; external;
```

There is, of course, no requirement that all methods be implemented as externals. Each individual method can be implemented in either Pascal or in assembly language, as desired.

The assembly language source file, RECT.ASM, that implements the three external methods is listed here.

```
TITLE Rect
LOCALS @@

; Rect structure
Rect STRUC
X1 DW 
Y1 DW 
X2 DW 
Y2 DW 
Rect ENDS

code SEGMENT BYTE PUBLIC

ASSUME cs:code

; procedure Rect.Init(XA, YA, XB, YB: Integer)

PUBLIC Rect@Init

Rect@Init PROC FAR

@XA EQU (WORD PTR [bp+16])
@YA EQU (WORD PTR [bp+14])
@XB EQU (WORD PTR [bp+12])
@YB EQU (WORD PTR [bp+10])
@Self EQU (DWORD PTR [bp+6])

push bp ;Save bp
mov bp,sp ;Set up stack frame
les di,@Self ;Load Self into ES:DI
cld ;Move forwards
mov ax,@XA ;X1 := XA
stosw
mov ax,@YA ;Y1 := YA
stosw
mov ax,@XB ;X2 := XB
stosw
mov ax,@YB ;Y2 := YB
stosw
pop bp ;Restore BP
ret 12 ;Pop parameters and return

Rect@Init ENDP

; procedure Rect.Union(var R: Rect)
PUBLIC Rect@Union

Rect@Union PROC FAR
@R EQU (DWORD PTR [bp+10])
@Self EQU (DWORD PTR [bp+6])
push bp ;Save BP
mov bp,sp ;Set up stack frame
push ds ;Save DS
lds si,@R ;Load R into DS:SI
les di,@Self ;Load Self into ES:DI
cld ;Move forward
lodsw ;If R.X1 >= X1 goto @@1
scasw
dec di ;X1 := R.X1
dec di
stosw
@@1: lodsw ;If R.Y1 >= Y1 goto @@2
scasw
dec di ;Y1 := R.Y1
dec di
stosw
@@2: lodsw ;If R.X2 <= X2 goto @@3
scasw
dec di ;X2 := R.X2
dec di
stosw
@@3: lodsw ;If R.Y2 <= Y2 goto @@4
scasw
dec di ;Y2 := R.Y2
dec di
stosw
@@4: pop ds ;Restore DS
Constructor error recovery

As described in Chapter 15 of the Reference Guide, Turbo Pascal allows you to install a heap error function through the HeapError variable in the System unit. This functionality is still supported in Turbo Pascal 5.5, but now it also affects the way object type constructors work.

By default, when there is not enough memory to allocate a dynamic instance of an object type, a constructor call using the extended syntax of the New standard procedure generates run-
time error 203. If you install a heap error function that returns 1 rather than the standard function result of 0, a constructor call through New will return nil when it cannot complete the request (instead of aborting the program).

The code that performs allocation and VMT field initialization of a dynamic instance is part of a constructor's entry sequence: When control arrives at the begin of the constructor's statement part, the instance will already have been allocated and initialized. If allocation fails, and if the heap error function returns 1, the constructor skips execution of the statement part and returns a nil pointer; thus, the pointer specified in the New construct that called the constructor is set to nil.

Once control arrives at the begin of a constructor's statement part, the object type instance is guaranteed to have been allocated and initialized successfully. However, the constructor itself might attempt to allocate dynamic variables, in order to initialize pointer fields in the instance, and these allocations might in turn fail. If that happens, a well-behaved constructor should reverse any successful allocations, and finally deallocate the object type instance so that the net result becomes a nil pointer. To make such "backing out" possible, Turbo Pascal implements a new standard procedure called Fail, which takes no parameters, and which can be called only from within a constructor. A call to Fail causes a constructor to deallocate the dynamic instance that was allocated upon entry to the constructor, and causes the return of a nil pointer to indicate its failure.

When dynamic instances are allocated through the extended syntax of New, a resulting value of nil in the specified pointer variable indicates that the operation failed. Unfortunately, there is no such pointer variable to inspect after the construction of a static instance or when an inherited constructor is called. Instead, Turbo Pascal allows a constructor to be used as a Boolean function in an expression: A return value of True indicates success, and a return value of False indicates failure due to a call to Fail within the constructor.

The following program implements two simple object types that contain pointers. This first version of the program does not implement constructor error recovery.

```pascal
type
  LinePtr = ^Line;
  Line = string[79];
```
BasePtr = ^Base;
Base = object
  L1, L2: LinePtr;
  constructor Init(S1, S2: Line);
  destructor Done; virtual;
  procedure Dump; virtual;
end;

DerivedPtr = ^Derived;
Derived = object(Base)
  L3, L4: LinePtr;
  constructor Init(S1, S2, S3, S4: Line);
  destructor Done; virtual;
  procedure Dump; virtual;
end;

var
  BP: BasePtr;
  DP: DerivedPtr;

constructor Base.Init(S1, S2: Line);
begin
  New(L1);
  New(L2);
  L1^ := S1;
  L2^ := S2;
end;

destructor Base.Done;
begin
  Dispose(L2);
  Dispose(L1);
end;

procedure Base.Dump;
begin
  WriteLn('B: ', L1^, ', ', L2^, '.');
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
  Base.Init(S1, S2);
  New(L3);
  New(L4);
  L3^ := S3;
  L4^ := S4;
end;

destructor Derived.Done;
begin
  Dispose(L4);
  Dispose(L3);
Base.Done;
end;

procedure Derived.Dump;
begin
  WriteLn('D: ', L1', ', ', L2', ', ', L3', ', ', L4', '.');
end;

begin
  New(BP, Init('Turbo', 'Pascal'));
  New(DP, Init('North', 'East', 'South', 'West'));
  BP^.Dump;
  DP^.Dump;
  Dispose(DP, Done);
  Dispose(BP, Done);
end.

The next example demonstrates how the previous one can be rewritten to implement error recovery. The type and variable declarations are not repeated, because they remain the same.

constructor Base.Init(S1, S2: Line);
begin
  New(L1);
  New(L2);
  if (L1 = nil) or (L2 = nil) then
  begin
    Base.Done;
    Fail;
  end;
  L1# := S1;
  L2# := S2;
end;

destructor Base.Done;
begin
  if L2 <> nil then Dispose(L2);
  if L1 <> nil then Dispose(L1);
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
  if not Base.Init(S1, S2) then Fail;
  New(L3);
  New(L4);
  if (L3 = nil) or (L4 = nil) then
  begin
    Derived.Done;
    Fail;
  end;
  L3# := S3;
L4 := S4;
end;

destructor Derived.Done;

begin
  if L4 <> nil then Dispose(L4);
  if L3 <> nil then Dispose(L3);
  Base.Done;
end;

{$F+}
function HeapFunc(Size: Word): Integer;
begin
  HeapFunc := 1;
end;
{$F-}

begin
  HeapError := @HeapFunc;  { Install heap error handler }
  New(BP, Init(’Turbo’, ’Pascal’));
  New(DP, Init(’North’, ’East’, ’South’, ’West’));
  if (BP = nil) or (DP = nil) then
    WriteLn(’Allocation error’)
  else
    begin
      BP^.Dump;
      DP^.Dump;
    end;
  if DP <> nil then Dispose(DP, Done);
  if BP <> nil then Dispose(BP, Done);
end.

Notice how the corresponding destructors in Base.Init and Derived.Init are used to reverse any successful allocations before Fail is called to finally fail the operation. Also notice that in Derived.Init, the call to Base.Init is coded within an expression so that the success of the inherited constructor can be tested.
New and modified error messages

The following compiler error messages have been modified or added in Turbo Pascal 5.5.

24 File components may not be files or objects
The component type of a file type cannot be an object type or a file type, or any structured type with an object type or file type component.

147 Object type expected.
The identifier does not denote an object type.

148 Local object types are not allowed.
Object types can be defined only in the outermost scope of a program or unit. Object type definitions within procedures and functions are not allowed.

149 VIRTUAL expected.
The keyword virtual is missing.

150 Method identifier expected.
The identifier does not denote a method.
151 Virtual constructors are not allowed.  
A constructor method must be static.

152 Constructor identifier expected.  
The identifier does not denote a constructor.

153 Destructor identifier expected.  
The identifier does not denote a destructor.

154 Fail only allowed within constructors.  
The *Fail* standard procedure can be used only within constructors.
INDEX

$ See compiler directives
@ (address operator)
  with method designators 80

A
activation, qualified 82
ancestors 9, 12, 68, 74
  assigning descendants to 32
  immediate 12
arrays
  range checking 71
assignment
  compatibility 77, 78, 81
  statements 81

B
/B command-line option
  in TINST or INSTALL 3
binding
  early 31
  late 31, 76
    Turbo Debugger and 65
    with polymorphic objects 38
Borland, contacting
  CompuServe 4
  mailing address 4
buffers
  overlay 89
    loading and freeing up 90
    optimization algorithm 90
  probationary area 91

C
C++ 8
calling conventions
  constructors and destructors 102

D
data
  objects
    changing 71
    inspecting 70, 71
  structures
    tracing through 71
debugger, integrated See methods, debugging;
  objects, debugging
debugging
  methods See methods, debugging
  objects See objects, debugging
declaration
  constructors 83
  destructors 83
  methods 15, 16, 76, 83
  object instances 13
  object types 74
Descend command (Turbo Debugger) 71
descendants 12, 68, 74
  immediate 12
designators
  field 19, 79
  method 78, 79
  @ (address operator) with 80
destructor (keyword) 73
destructors 84
  calling conventions 102
  declaring 52, 83
  defined 52, 84
  dynamic object disposal 54
  implementation 83
  polymorphic objects and 53
  static versus virtual 52
directives See compiler directives
Dispose procedure
  extended syntax 51, 99, 103
    constructor passed as parameter 79, 84, 86
domain, object 74
dotting 13, 17, 21
dynamic object instances 49-60
  allocation and disposal 50, 54, 102

E
early binding 31
encapsulation 9, 23
error checking
  dynamic object allocation 106
  virtual method calls 99
error messages
  compiler 111
  Turbo Debugger 72
Evaluate window
  calling methods in 67
  objects and 64, 66

event handling
  virtual methods and 42
exported object types 19
extensibility 47, 78
external (keyword) 103

F
fields, object 13, 74
  accessing 14, 16, 23
  designators 19, 79
  inherited 13
  scope 17, 76, 83
  method parameters and 19
files
  graphics, installing 3
  obsolete, deleting 3
Find Procedure command
  methods and 64
format specifiers
  objects 64
Function Inspector window (Turbo Debugger) 70
functions
  heap error 106
  methods denoting 81
  OvrGetRetry 91, 95
  SizeOf 100
  TypeOf 101

G
GRAPH.TPU
  installed in a separate directory 3
graphics
  files, installed in a separate directory 3

H
heap error function 106
hierarchies
  object 12
    common attributes in 38, 41
    tree 68
Hierarchy command (Turbo Debugger) 70, 72

I
IDE See integrated development environment
immediate ancestors and descendants 12
implementation
  constructors 83
  destructors 83
  methods 76, 83
inheritance 9, 10, 11, 74
  showing during debugging 70, 71
InitGraph procedure
  path name to graphics directory 3
Inspect command (Turbo Debugger) 68, 69, 70, 71
INSTALL 2
  /B command-line option 3
LCD or composite screen display
  adjusting 3
installing Turbo Pascal 2
instances
  defined 11
  dynamic object 49-60
  object 77
    declaring 13
    linked lists of 54
  static object 10-49
integrated debugger See methods, debugging; objects, debugging

K
keywords
  case 82
  constructor 36, 73
  destructor 73
  external 103
  object 12, 73
  virtual 35, 73, 76
  with 79, 82

L
$L$ compiler directive 103
laptop computers
  display, adjusting 3
late binding 31, 76
  Turbo Debugger and 65
  with polymorphic objects 38
LCD mode
  display, adjusting 3
linked lists 54

M
methods See also objects
  activation, qualified 82
assembly language 18, 103
calling 15
  as functions or procedures 81
  conventions 81, 102
debugging 63, 65
declaring 15, 16, 83
defined 14, 74
designators 78, 79
  @ (address operator) with 80
external 18, 103
Find Procedure command and 64
Function Inspector window 70
identifiers, qualified 76
  accessing object fields 21, 79
  in method calls 78, 82
  in method declarations 15, 17, 83
  scope and 65
implementation 76, 83
inspecting 69-70, 71
overridden, calling 82
overriding inherited 25, 77
parameters
  naming 19
  Self 18, 81, 82, 83
    debugging and 64, 66
    defined 102
    explicit use of 18
  type compatibility 86
positioning in hierarchy 41
procedures versus 41
qualified activation 82
scope 17
static 29, 76
  calling 78
  problems with inherited 27
virtual 30, 76
  calling 78, 81, 101
    error checking 99
    event handling and 42
    polymorphic objects and 35
    static versus 41
methods, declaring 76
Methods command (Turbo Debugger) 71
New Expression command (Turbo Debugger) 71
New procedure 50
   extended syntax 50, 99
      constructor passed as parameter 79, 84, 86, 102
      used as function 51, 87

object (keyword) 12, 73
Object Hierarchy window (Turbo Debugger) 70, 67-72
Object Instance Inspector window (Turbo Debugger) 70
      array ranges 71
      changing data values 71
      methods and 71
Object Type Inspector window (Turbo Debugger) 68, 69
      complex data structures and 71
objects  See also methods
   ancestor 12, 74
   constructors
      declaring 83
      defined 36, 84
      error recovery 106
      implementation 83
      inherited 77
      virtual methods and 36
      VMTP and 37, 50, 79, 84, 97, 100
data
   changing 71
   inspecting 70, 71
debugging 63
   Evaluate window and 64
   stepping and tracing 63
   Watch window and 64
defined 8
descendant 12, 74
destructors 84
   declaring 52, 83
   defined 52, 84
dynamic object disposal 54
   implementation 83
   polymorphic objects and 53

static versus virtual 52
domain 74
dynamic instances 49-60
   allocation and disposal 50, 54, 79, 84, 102
extensibility 47
fields 13, 74
   accessing 14, 16, 23
   designators 19, 79
   inherited 13
   scope 17, 76, 83
   method parameters and 19
hiding data representation 24
hierarchies 12
   common attributes in 38, 41
   inspecting 67
   tree 68
inheritance 74
   showing during debugging 70, 71
inspecting 69-70
instances 77
   declaring 13
   linked lists of 54
internal data format 97
passed as parameters
   compatibility 33
   pointers to 77
   compatibility 32
polymorphic 33, 77, 78, 81, 86
   late binding and 38
relative position 61
static instances 10-49
Turbo Debugger and 65
typed constants of type 80
types 74
   compatibility 31
   exported by units 19
   inspecting 69-70
   list of 68
types. declaring 74
units and 19
virtual method table 99
   pointer 97
      initialization 37, 100
virtual methods
   call error checking 99
   calling 101
Overlay unit 89
  procedures and functions 91, 94
  variables 91
overlays
  buffer 89
    loading and freeing up 90
    optimization algorithm 90
    probationary area 91
    in .EXE files 95
    load operations, customizing 92
manager 89
overridden methods, calling 82
overriding inherited methods 25, 77
OvrGetRetry function 91, 95
OvrSetRetry procedure 91, 94

P
parameters
  method, naming 19
  Self 18, 81, 82, 83
    debuging and 64, 66
    defined 102
    explicit use of 18
    type compatibility 86
VMT 102
pointers
  assignment compatibility 77
  to objects 77
polymorphic objects 33
  late binding and 38
  virtual methods and 35
polymorphism 30, 31, 32, 33
  assignment compatibility 78
  object instance assignment 81
  parameter type compatibility 86
  pointer assignment 77
probationary area, overlay buffer 91
procedures
  Dispose
    extended syntax 51, 99, 103
      constructor passed as parameter 79, 84,
      86, 102
    used as function 51, 87
OvrSetRetry 91, 94
  statements 81
programs
  debugging  See debugging
Q
qualified activation 82
qualified method identifiers 76
  accessing object fields 21, 79
  in method calls 78, 82
  in method declarations 15, 83
  scope and 65
R
$R compiler directive
  virtual method checking 37, 99
Range command (Turbo Debugger) 71
records
  types 11
  relative position 61
S
scope
  object fields and methods 65
  scope, object fields and methods 17
Self parameter 18, 81, 82, 83
  debugging and 64, 66
  defined 102
  explicit use of 18
Show Inherited command (Turbo Debugger) 70,
  71
Simula-67 23
simulations, computerized 23
SizeOf function 100
Smalltalk 8, 23
statements
  assignment 81
  case 82
  procedure 81
  with 13, 22, 79, 82
    implicit 17
static methods 29, 76
  calling 78
problems with scope of inherited 27
static object instances 10-49
Step Over command
methods and 63, 65

T

taxonomy 10
technical support 4, 5
TINST
   /B command-line option 3
   LCD or composite screen display
      adjusting 3
Trace Into command
   methods and 63, 65
Tree command (Turbo Debugger) 68
trees
   object hierarchy 68
Turbo Debugger  See also methods, debugging;
   objects, debugging
   Change command 71
   Descend command 71
   Function Inspector window 70
   Hierarchy command 70, 72
   Inspect command 68, 69, 70, 71
   Methods command 71
   New Expression command 71
   Object Hierarchy window 70, 69-72
   Object Instance Inspector window 70
   Object Type Inspector window 68, 69
   objects and  See objects
   Range command 71
   Show Inherited command 70, 71
   Tree command 68
Turbo Pascal
   installing 2
typed constants
   object type 80
TypeOf function 101
types
   object 74

exported by units 19
object. declaring 74
record 11

U

/U command-line option
   GRAPH.TPU file and 3
Unit Directories command
   GRAPH.TPU file and 3
units
   objects in 19
   Overlay 89
utilities  See INSTALL; TINST

V

VER55 symbol 87
virtual (keyword) 35, 73, 76
virtual method table 37, 99
   pointer 97
      initialization 37, 100
virtual methods 30, 76
   calling 78, 81, 101
      error checking 99
   event handling and 42
   polymorphic objects and 35
   static versus 41
VMT  See virtual method table
VMT parameter 102
VMTP  See virtual method table pointer

W

Watch
   window
      objects and 64
with (keyword)
   statement 13, 22, 79, 82
      implicit 17