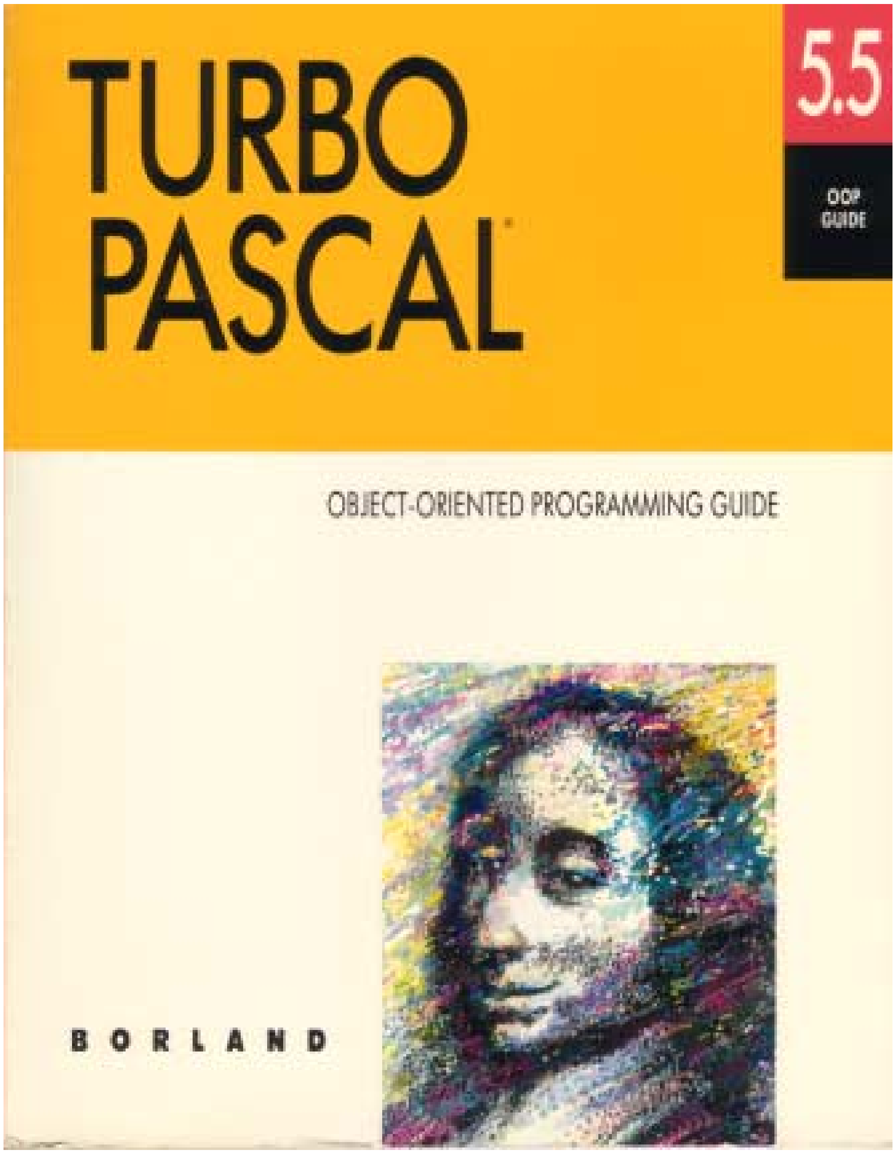

TURBO PASCAL®

5.5

OOP GUIDE

OBJECT-ORIENTED PROGRAMMING GUIDE

BORLAND

Turbo Pascal 5.5

Object-Oriented Programming Guide

Disclaimer:
This volume is not the complete text of the original book, and has been scanned from the original, converted through OCR, and errors have been corrected by hand. Any remaining errors are mine, overlooked in this process, despite efforts to find and correct them.

I undertook this project because of repeatedly expressed wishes by many on the Borland newsgroups to see this material made available again. Borland kindly granted permission to republish, and this is the result. I hope that it may be of benefit to still more developers than it was in its original incarnation.
- William Meyer

Introduction

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,
  ```
  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 **O**ptions/**D**irectories/**U**nit 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
`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 mono- chrome/ 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

Technical Support Department
Borland International
P.O. Box 660001
1800 Green Hills Road
Scotts Valley, CA 95066-0001

Note!

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

CHAPTER 1

All about OOP

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).

Objects keep all their characteristics and behavior together.

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).

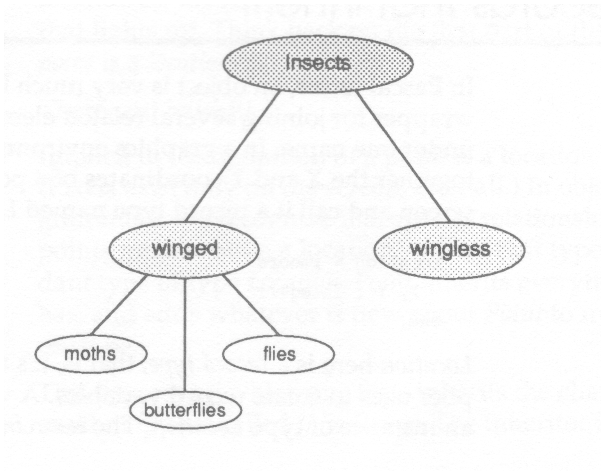

Figure 1.1
A partial taxonomy chart of insects

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:

```
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 descendant 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:

Note the use of parentheses here to denote inheritance.

```
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 inheritance. You can speak about *Point's X* value, just as you can speak about *Location's X* value.

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:

```
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,

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

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

You will just have to remember at first (it will eventually come naturally) that inherited fields are just as accessible as fields *Y* 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*:

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

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 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?

An object's data fields are what an object knows; its methods are what an object does.

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:

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

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

```
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:

```
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:

```
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:

```
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:

```
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:

```
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.

Method scope and the Self parameter

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 `Get Y: = 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.

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:

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.

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

Is if 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.

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

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;
```

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 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.

Object data fields and method formal parameters

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:

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

A procedure's loca1 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.

Objects exported by units

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.

Exported means "defined within the interface section of 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:

```
unit Points;
interface
uses Graph;
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 NoveTo(NewX, NewY ; Integer);
  end;
implementation
{---------------------------------------------------------}
{ Location's method implementations:                       }
{---------------------------------------------------------}
  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;

{---------------------------------------------------------}
{ Points's method implementations:                         }
{---------------------------------------------------------}
  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.NoveTo(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.NoveTo(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;

```
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.

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.

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

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:

```
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;
```

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:

```
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.

Methods: no downside

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.

A note about data abstraction

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.

Extending objects

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 }
NriteLn(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 NoveTo(NewX, NewY : Integer);
    procedure Contract(ContractHy : 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.Contract(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 if 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.

Important!

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 info 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*:

```
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 *Poinf.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.

Virtual methods and polymorphism

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.

IMPORTANT!

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 provides 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:

```
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:

An ancestor object can be assigned an instance of any of its descendant types.

```
  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,

```
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.

Warning! 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,

```
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:

```
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 designers 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:

```
type
  Location = object X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    function GetX : Integer;
    function GetY : Integer;
  end;
  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); viztua1;
    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.

Every object type that has virtual methods must have a constructor.

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

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.

Warning!

Each individual instance of an object must be initialized by a separate constructor call. If 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 constructor does is establish a link between the instance calling the constructor and the object type's VMT.

That’s important to remember: There is only one virtual method table for each object type. Individual instances of an object type (that is, variables of that type) contain a link to the VMT - they do not contain the VMT itself. The constructor sets the value of that link to the VMT - which is why you can launch execution into nowhere by calling a virtual method before calling the constructor.

Range checking virtual-method calls

The default state of $R is inactive. ($R-).

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.

Once virtual, always virtual

You'll notice that both *Point* and *Circ1e* have methods named *Show* and *Hide*. All method headers for *Show* and *Hide* are tagged as virtual methods with the reserved word virtual. Once an ancestor object type tags a method as virtual, all its descendant types that implement a method of that name must tag that method 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 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 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.

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.

A note about abstract objects

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:

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

```
procedure DragIt(var AnyFigure: Point; DragBy: Integer);
var
  DeltaX, Delta Y: Integer; FigureX,FigureY : Integer;
begin
  AnyFigure.Show;            { Display figure to be dragged }
  FigureX := AnyFigure.GetX; { Get the initial X,Y of figure }
  FigureY := AnyFigure.GetY;
  { 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:

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

```
unit Figures; { Virtual methods 6 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; vt.rtual;
    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 : Inteqer); virtual;
  end;
implementation
{--------------------------------------------------------}
{ Location's method implementations:                     }
```

```
{-------------------------------------------------------}
  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;

{-------------------------------------------------------}
{ Point's method implementations:                       }
{-------------------------------------------------------}
  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.NoveTo(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 }
             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;
end;

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

```
      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;
    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.Contract(ContractBy : Integer);
  begin
    Expand(-ContractBy);
  end;

  { No initialization section }
end.
```

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.

Object extensibility

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 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:

```
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;
  GraphNode : 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;
  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;

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Main program:
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
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 (1);
  end;
  { All descendants of type Point contain virtual methods and }
  { *must* be initialized before use through a constructor call.}
  ACircle.Init(151, 82,   { Initial X,Y at 151,82 }
               50);       { Initial radius of 50 pixels }
  AnArc.Init(151, 82,     { Initial X,Y at 151,82 }
             25,          { Initial radius of 50 pixels }
             0, 90);      { 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.
```

Static or virtual methods

In general, you should make methods virtual. Use static methods only when you want to optimize for speed and memory efficiency. The trade-off, 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, *Paint* 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 such 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.

```
var
  ACircle : Circle;
```

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:

```
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:

```
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:

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

Allocation and initialization with New

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:

```
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 compiler 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
  PArc : ArcPtr;

PArc := 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 new standard procedure, Fail, helps you do error recovery in construction; see page 107.

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

Disposing dynamic objects

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 dean 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:

```
MyComplexObject.Done;
```

We suggest the identities 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:

```
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 &om 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:

```
  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:

```
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.

An example of dynamic object allocation

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*.

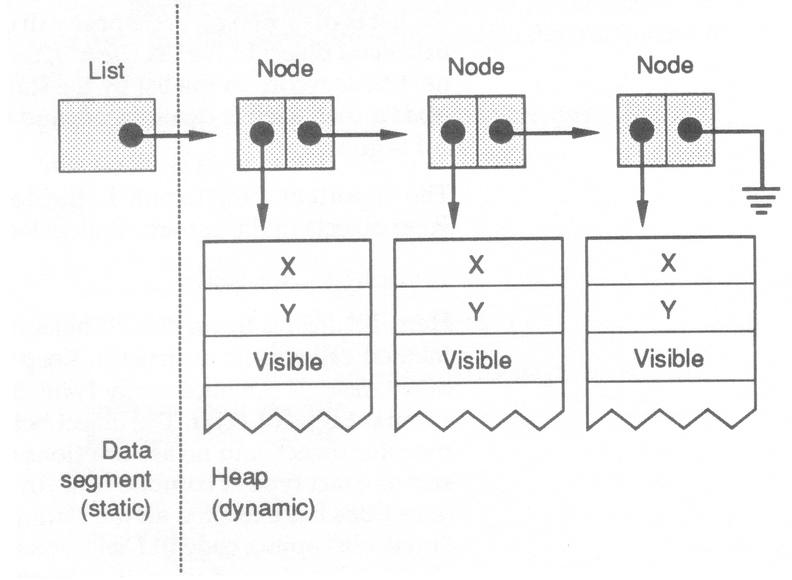

Fig 1.2

Disposing of a complex data structure on the heap

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:

```
  AList.Done;
```

The code for the destructor merits examination:

```
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:

```
  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, *AList* is declared as a static variable in the data segment. *AList* could as easily have been itself allocated on the heap, and anchored to reality by a pointer of type *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 *Dispose*:

```
var PList : ListPtr;
...
  Dispose(PList,Done);
```

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

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

```
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;
    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: }
constructor List.Init;
begin
  Nodes := nil;
end;

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;

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

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

```
end;

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Main program:                                              }
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
begin
  { Let the BGI determine what board you're using: }
  InitGraph(GraphDriver, GraphNode, '');
  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. Inheritance 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.