

# THOSE MAGIC TIES, DBM STUFF, AND THE DATABASE HOOKS

## 14.1 Tying Variables to a Class

Normally when you perform some operation on a variable, such as assigning, changing, or printing the value of the variable, Perl performs the necessary operations on that variable internally. For example, you don't need a constructor method just to create a variable and assign a value to it, and you don't have to create access methods to manipulate the variable. The assignment statement `$x=5;` doesn't require any tricky semantics. Perl creates the memory location for `$x` and puts the value 5 in that location.

It is now possible to bind an ordinary variable to a class and provide methods for that variable so that, as if by magic, the variable is transformed when an assignment is made or a value is retrieved from it. A scalar, array, or hash, then, can be given a new implementation. Unlike objects where you must use a reference to the object, tied variables, once created, are treated like any other variable. All of the details are hidden from the user. You will use the same syntax to assign values to the variable and to access the variable as you did before tying.<sup>1</sup> The magic goes on behind the scenes. Perl creates an object to represent the variable and uses predefined method names to construct, set, get, and destroy the object that has been tied to the variable. The programmer who creates the class uses the predefined method names, such as *FETCH* and *STORE*, to include the statements necessary to manipulate the object. The user ties the variable and, from that point on, uses it the same way as he would any other variable in his program.

### 14.1.1 The *tie* Function

The *tie* function binds a variable to a package or class and returns a reference to an object. All the details are handled internally. The *tie* function is most commonly used with associative arrays to bind key/value pairs to a database; for example, the DBM modules pro-

---

1. DBM databases use the *tie* mechanism to automatically perform database operations on data.

vided with the Perl distribution use tied variables. The *untie* function will disassociate a variable from the class to which it was tied. The format for *tie* follows.

## FORMAT

```
$object = tie variable, class, list;
untie variable;

tie variable, class, list;
$object = tied variable;
```

The *tie* function returns a reference to the object that was previously bound with the *tie* function or undefined if the variable is not tied to a package.

### 14.1.2 Predefined Methods

Tying variables allows you to define the behavior of a variable by constructing a class that has special methods to create and access the variable. The methods will be called automatically when the variable is used. A variable can't be tied to any class, but must be tied to a class that has predefined method names. The behavior of the variable is determined by methods in the class that will be called automatically when the variable is used. The constructors and methods used to tie (constructor) and manipulate (access methods) the variable have predefined names. The methods will be called automatically when the tied variables are fetched, stored, destroyed, and so on. All of the details are handled internally. The constructor can bless and return a pointer to any type of object. For example, the reference may point to a blessed scalar, array, or hash. But the access methods **must return** a scalar value if *TIESCALAR* is used, an array value if *TIEARRAY* is used, and a hash value if *TIEHASH* is used.

### 14.1.3 Tying a Scalar

In order to use a tied scalar, the class must define a set of methods that have predefined names. The constructor, *TIESCALAR*, is called when the variable is tied and it creates the underlying object that will be manipulated by the access methods, *STORE* and *FETCH*. Any time the user makes an assignment to the tied scalar, the *STORE* method is called and whenever he attempts to display the tied scalar, the *FETCH* method is called. The *DESTROY* method is not required, but if it is defined, will be called when the tied scalar is untied or goes out of scope. Methods provided for a tied scalar are

```
TIESCALAR $classname, LIST
STORE $self, $value
FETCH $self
DESTROY $self
```



**EXAMPLE 14.1 (CONTINUED)**

```

(Output)
14 object is Square=SCALAR(0x1a72da8) .
15 25
16 625
   390625
-----
18 9
   81
   6561

```

**EXPLANATION**

- 1 The package/class *Square* is declared. The file is *Square.pm*.
- 2 *TIESCALAR* is the constructor for the class. It creates an association between the tied scalar and an object. Look at line 13 of this example. This is where the constructor is called. The variable tied to the object is *\$squared*.
- 3 The first argument to the constructor is the name of the class. It is shifted from the *@\_* array and assigned to *\$class*.
- 4 Look again at line 13. The tied variable, *\$squared*, is followed by the name of the class, and then the number 5. The class name is passed first, followed by 5. In the constructor the number 5 is shifted from the *@\_* array and assigned to *\$data*.
- 5 A reference to the scalar is created and passed to the *bless* function, which creates the object.
- 6 The *FETCH* method is defined. This access method will retrieve data from the object.
- 7 The first argument to the *FETCH* method is a reference to the object.
- 8 The pointer is de-referenced and its value squared.
- 9 The *STORE* method is defined. It will be used to assign a value to the object.
- 10 The first argument to the *STORE* method is a reference to the object.
- 11 The value to be assigned is shifted off the *@\_* array and assigned to *\$\$self*.
- 12 This is the user/driver program. The *Square* module is loaded into the program.
- 13 The *tie* function automatically calls the constructor *TIESCALAR*. The *tie* function ties the variable, *\$squared*, to the class and returns a reference to the newly created object.
- 14 The reference, *\$object*, points to a scalar variable found in the *Square* class.
- 15 The *FETCH* method is automatically called and the current value of *\$squared* is printed.
- 16 The *FETCH* method is called again. Each time the method is called the current value of the tied variable is squared again and returned.
- 17 The *STORE* method is automatically called. A new value, 3, is assigned to the tied variable.
- 18 The *FETCH* method is automatically called and displays the squared value.
- 19 The *untie* method disassociates the object with the tied variable. If a *DESTROY* method had been defined, it would have been called at this point.

Example 14.2 also ties a scalar, *\$cost*, and demonstrates how to access the tied scalar.

### EXAMPLE 14.2

```

# File: Markup.pm
1  package Markup;
2  sub TIESCALAR { # Constructor ties a scalar
3      my $class = shift;
4      my $markup = shift;
5      my $cost = 0;
6      my $mcost = [ $markup, $cost ];
7          # Create a reference to an anonymous array
8      return bless ( $mcost, $class); # Bless the reference
9  }

10 sub STORE {
11     my $self = shift;
12         # First argument is a reference to the object
13     my $amount = shift;
14         # Second argument is the value assigned to $cost
15     $self->[1] = $amount * $self->[0]; # Returns a scalar
16 }

17 sub FETCH {
18     my $self = shift;
19         # First argument is a reference to the object
20     return $self->[1];
21         # The pointer is dereferenced; returns a scalar
22 }

23 sub DESTROY {
24     print "The object is being destroyed.\n";
25 }
26 1;

-----

# This is the user/driver program
27 use Markup;
28 $object = tie $cost, 'Markup', 1.15;
29     # Ties the variable $cost to the Markup package
30     # Whenever the variable $cost is accessed, it will be passed
31     # to the FETCH and STORE methods and its value manipulated by
32     # the object it is tied to. A scalar must be returned by
33     # FETCH and STORE if tying to a scalar. The constructor
34     # can bless a reference to any data type, not specifically
35     # a scalar.
36 print "What is the price of the material? ";
37 $cost = <STDIN>;
38     # Could have said: $object->STORE(34)
39     # or could have said: (tied $cost)->STORE(34);

```

**EXAMPLE 14.2 (CONTINUED)**

```

19 printf "The cost to you after markup is %.2f.\n", $cost;
    # Could have said: $object->FETCH
20 untie $cost; # $cost is now a normal scalar
    # The DESTROY method is called automatically when
    # the program exits

```

```

-----

(Output)
17 What is the price of the material? 34
19 The cost to you after markup is 39.10.
20 The object is being destroyed.

```

**EXPLANATION**

- 1 A package (class) called *Markup* is declared.
- 2 A predefined method name called *TIESCALAR* is defined and will act as the constructor; it is automatically invoked when the variable is tied.
- 3 The parameters being passed to the constructor are assigned to the `@_` array. See line 13 in the driver program. The first argument shifted off is the name of the class called *Markup*.
- 4 The second argument shifted off is the value *1.15*.
- 5 The reference *\$mcost* is assigned the address of an anonymous array consisting of two values, *\$mcost* and *\$class*.
- 6 The reference is blessed into the class. Note that it doesn't matter whether the reference blessed is to a hash, array, or scalar. The *tie* function only requires that the **access** methods return a scalar if a variable is tied to a scalar as shown in line 13, an array if the variable is tied to an array, and a hash if the variable is tied to a hash. In this example, the value that will be returned by the access methods, *FETCH* and *STORE*, must be a scalar, and they are.
- 7 *STORE* is an access method used to set or assign values for an object. The name *STORE* is predefined for tied variables.
- 8 The first argument to the *STORE* method is a reference to the object. It is shifted from the `@_` array and assigned to *\$self*.
- 9 The second argument shifted off is the amount spent on the material. (See line 18.)
- 10 Remember, the reference *\$self* points to an anonymous array (the object) that contains two values: the markup cost, and the actual cost of the material. The cost of the material was initialized to *0* in the constructor, *TIESCALAR*. The amount, *\$amount*, passed to the *STORE* method is multiplied by the markup price, *\$self->[0]*. The result is assigned to *\$self->[1]*, the cost of the material after the markup. The tied object has been modified. On line 15 in the driver program, it appears that the only assignment is coming from *STDIN*. As soon as the assignment statement is executed, the *STORE* method is automatically called, behind the user's back, so to speak.

**EXPLANATION (CONTINUED)**

- 11 *FETCH* is an access method used to get or retrieve data from an object. The name *FETCH* is predefined for tied variables.
- 12 The first argument to the *FETCH* method is a reference to the object. It is shifted from the *@\_array* and assigned to *\$self*.
- 13 The *FETCH* method retrieves and returns the value of the cost after the markup.
- 14 The *DESTROY* method will be called just before the object is removed, either when it goes out of scope or when the program ends.
- 15 The *Markup* module is loaded into the user's program.
- 16 The scalar *\$cost* is tied to the *Markup* class. When *TIESCALAR*, the constructor, is called, it will get the name of the class, *Markup*, and 1.15. *\$object* is a reference to the object that is tied.
- 17, 18 The user is asked for input. The *STORE* method is automatically called.
- 19 *FETCH* is called and the value of *\$cost* is printed.
- 20 The object is destroyed with *untie*.

**14.1.4 Tying an Array**

In order to use a tied array, the class must define a set of methods that have predefined names. The constructor, *TIEARRAY*, is called when the variable is tied and it creates the underlying object that will be manipulated by the access methods, *STORE* and *FETCH*. Any time the user makes an assignment to the tied array, the *STORE* method is called, and whenever he attempts to display the tied array, the *FETCH* method is called. The *DESTROY* method is not required, but, if it is defined, will be called when the tied scalar is untied or goes out of scope. There are also a set of optional methods that can be used with tied arrays. *STORESIZE* sets the total number of items in the array, *FETCHSIZE* is the same as using *scalar(@array)* or *\$#array + 1* to get the size of the array, and *CLEAR* is used when the array is to be emptied of all its elements. There are also a number of methods, such as *POP* and *PUSH*, that emulate their like-named Perl functions in manipulating the array. Methods provided for an array are

```
TIEARRAY $classname, LIST
STORE $self, $subscript, $value
FETCH $self, $subscript, $value
DESTROY $self
STORESIZE $self, $arraysize
FETCHSIZE $self
EXTEND $self, $arraysize
EXISTS $subscript
DELETE $self, $subscript
CLEAR $self
PUSH $self, LIST
```

```

UNSHIFT $self, LIST
POP $self
SHIFT $self
SPLICE $self, OFFSET, LENGTH, LIST

```

There is also a base class called *Tie::Array* in the standard Perl library that contains a number of predefined methods from the above list, making the implementation of tied arrays much easier. To see documentation for this module, type *perldoc Tie::Array*.

### EXAMPLE 14.3

```

1  package Temp;
2  sub TIEARRAY {
3      my $class = shift; # Shifting the @_ array
4      my $obj = [ ];
5      bless ($obj, $class);
6  }
7  # Access methods
8  sub FETCH {
9      my $self=shift;
10     my $indx = shift;
11     return $self->[$indx];
12 }
13
14 sub STORE {
15     my $self = shift;
16     my $indx= shift;
17     my $F = shift; # The Fahrenheit temperature
18     $self->[$indx]=$F - 32) / 1.8; # Magic works here!
19 }
20
21 1;
-----
#!/bin/perl
# The user/driver program
22 use Temp;
23 tie @list, "Temp";
24 print "Beginning Fahrenheit: ";
25 chomp($bf = <STDIN>);
26     print "Ending temp: ";
27     chomp($ef = <STDIN>);
28 print "Increment value: ";
29 chomp($ic = <STDIN>);
30 print "\n";
31 print "\tConversion Table\n";
32 print "\t-----\n";

```

**EXAMPLE 14.3 (CONTINUED)**

```

18 for($i=$bf;$i<=$ef;$i+=$ic){
19     $list[$i]=$i;
20     printf"\t$i F. = %.2f C.\n", $list[$i]; }

```

(Output)

```

17 Beginning Fahrenheit: 32
    Ending temp: 100
    Increment value: 5

```

Conversion Table

```

-----
20 2 F. = 0.00 C.
    37 F. = 2.78 C.
    42 F. = 5.56 C.
    47 F. = 8.33 C.
    52 F. = 11.11 C.
    57 F. = 13.89 C.
    62 F. = 16.67 C.
    67 F. = 19.44 C.
    72 F. = 22.22 C.
    77 F. = 25.00 C.
    82 F. = 27.78 C.
    87 F. = 30.56 C.
    92 F. = 33.33 C.
    97 F. = 36.11 C.

```

**EXPLANATION**

- 1 This is the package/class declaration. The file is *Temp.pm*.
- 2 *TIEARRAY* is the constructor for the tied array. It creates the underlying object.
- 3 The first argument passed into the constructor is the name of the class, *Temp*.
- 4 A reference to an anonymous array is created.
- 5 The object is blessed into the class.
- 6 The access method, *FETCH*, will retrieve elements from the tied array.
- 7 The first argument shifted off and assigned to *\$self* is a reference to the object.
- 8 The next argument is the value of the subscript in the tied array. It is shifted off and assigned to *\$indx*.
- 9 The value of the array element is returned when the user/driver program attempts to display an element of the tied array.
- 10 The access method, *STORE*, will assign values to the tied array.
- 11 The first argument shifted off and assigned to *\$self* is a reference to the object.
- 12 The next argument is the value of the subscript in the tied array. It is shifted off and assigned to *\$indx*.

**EXPLANATION (CONTINUED)**

- 13 The value of the Fahrenheit temperature is shifted from the argument list. See line 19 where the assignment is being made in the user/driver program. The value being assigned is what will be stored in *\$F* in the *STORE* method.
- 14 The calculation to convert from Fahrenheit to Celsius is made on the incoming tied array element. The user/driver program never sees this calculation, as though it were done by magic.
- 15 This is the user's program. The *Temp.pm* module is loaded into the program.
- 16 The *tie* function ties the array to the class, *Temp* and returns an underlying reference to an object that is tied to the array.
- 17 The user is asked for input. He will provide a beginning Fahrenheit temperature, an ending Fahrenheit temperature, and an increment value.
- 18 A *for* loop is entered to iterate through the list of temperatures. It charts the Fahrenheit temperature and the corresponding Celsius temperature after the conversion.
- 19 The magic happens here. When the assignment is made, the *STORE* method is automatically called, where the formula converts the Fahrenheit temperature to Celsius and assigns it to the array. A reference to the tied object and the array index are passed to *STORE*.
- 20 When *printf* is used, the *FETCH* method will automatically be called and display an element of the tied array. A reference to the tied object and the array index are passed to *FETCH*.

**14.1.5 Tying a Hash**

In order to use a tied hash, the class must define a set of methods that have predefined names. The constructor, *TIEHASH*, is called when the variable is tied and it creates the underlying object that will be manipulated by the access methods, *STORE* and *FETCH*. Any time the user makes an assignment to the tied hash, the *STORE* method is called, and whenever he attempts to display the tied hash, the *FETCH* method is called. The *DESTROY* method is not required, but, if it is defined, will be called when the tied hash is untied or goes out of scope. There are also a set of optional methods that can be used with tied hashes. *DELETE* removes a key/value pair, *EXISTS* checks for the existence of a key, and *CLEAR* empties the entire hash. If you use Perl's built-in keys, values, or *each* methods, the *FIRSTKEY* and *NEXTKEY* methods are called to iterate over the hash. Methods provided for an associative array are

```
TIEHASH $classname, LIST
FETCH $self, $key
STORE $self, $key
DELETE $self, $key
EXISTS $self, $key
FIRSTKEY $self
NEXTKEY $self, $lastkey
```

```
DESTROY $self
CLEAR $self
```

There is also a base class module called *Tie::Hash* in the standard Perl library that contains a number of predefined methods from the above list, making the implementation of tied arrays much easier. To see documentation for this module, type *perldoc Tie::Hash*.

### EXAMPLE 14.4

```
(The Script)
#!/bin/perl
# Example using tie with a hash
1 package House;
2 sub TIEHASH {          # Constructor method
3     my $class = shift;    # Shifting the @_ array
4     my $price = shift;
5     my $color = shift;
6     my $rooms = shift;
7     print "I'm the constructor in class $class.\n";
8     my $house = { Color=>$color,    # Data for the tied hash
9                 Price=>$price,
10                Rooms=>$rooms,
11                };
12    bless $house, $class;
13 }
14 sub FETCH {           # Access methods
15     my $self=shift;
16     my $key=shift;
17     print "Fetching a value.\n";
18     return $self->{$key};
19 }
20 sub STORE {
21     my $self = shift;
22     my $key = shift;
23     my $value = shift;
24     print "Storing a value.\n";
25     $self->{$key}=$value;
26 }
27 1;
```

---

**EXAMPLE 14.4 (CONTINUED)**

```

# User/driver program
13 use House;
# The arguments following the package name are
# are passed as a list to the tied hash
# Usage: tie hash, package, argument list
# The hash %home is tied to the package House.
14 tie %home, "House", 155000, "Yellow", 9;
# Calls the TIEHASH constructor
15 print qq/The original color of the house: $home{"Color"}\n/;
# Calls FETCH method
16 print qq/The number of rooms in the house: $home{"Rooms"}\n/;
17 print qq/The price of the house is: $home{"Price"}\n/;
18 $home{"Color"}="beige with white trim"; # Calls STORE method
19 print "The house has been painted. It is now $home{Color}.\n";
20 untie(%home); # Removes the object

```

(Output)

```

4 I'm the constructor in class House.
9 Fetching a value.
15 The original color of the house: Yellow
9 Fetching a value.
16 The number of rooms in the house: 9
9 Fetching a value.
17 The price of the house is: 155000
Storing a value.
Fetching a value.
The house has been painted. It is now beige with white trim.

```

**EXPLANATION**

- 1 The package/class *House* is declared. The file is *House.pm*.
- 2 The constructor *TIEHASH* will tie a hash to an object.
- 3 The first argument is the name of the class, *\$class*. The rest of the arguments shifted and all will be assigned as values to the keys of an anonymous hash, the object.
- 4 The printout shows that the constructor class is called *House*.
- 5 A reference to an anonymous hash is created and with key/value pairs assigned. They will become the properties of the object.
- 6 The values assigned to the keys of the anonymous hash were passed to the constructor, *TIEHASH*. (See line 13 in the user program.)
- 7 The  *bless*  function returns a reference to the object that is created.
- 8 The  *FETCH*  method is an access method that will retrieve a value from the hash object.
- 9 Each time the user uses the  *print*  function to display a value from the hash, the  *FETCH*  method is automatically called and this line will be printed.
- 10 A value from the hash is returned.

**EXPLANATION (CONTINUED)**

- 11 The *STORE* method is an access method that will be called automatically when the user attempts to assign a value to one of the keys in the hash.
- 12 The value to be assigned to the hash came from the argument list passed to the *STORE* method.
- 13 This is the user/driver program. The *House* module is loaded into the program.
- 14 The *tie* function calls the *TIEHASH* constructor in *House.pm*. The hash, *%home*, is tied to an object in the *House* class. The name of the class, *House*, and three additional arguments are passed.
- 15 This line causes the *FETCH* access method to be called which will display the value for the hash key, *Color*.
- 16 This line causes the *FETCH* access method to be called which will display the value for the hash key, *Rooms*.
- 17 This line causes the *FETCH* access method to be called which will display the value for the hash key, *Price*.
- 18 This line causes the *STORE* access method to be called automatically when a value is assigned to one of the hash keys, in this case the key *Color*.
- 19 The *print* function causes the *FETCH* method to be called automatically to display a value for a specified hash key, *Color*.
- 20 The *untie* function disassociates the hash from the object to which it was tied.

**EXAMPLE 14.5**

```

# File is House.pm
1 package House;
2 sub TIEHASH {
    my $class = shift;
    print "I'm the constructor in package $class\n";
    my $housereref = {};
    bless $housereref, $class;
}
3 sub FETCH {
    my $self=shift;
    my $key=shift;
    return $self->{$key};
}
4 sub STORE {
    my $self = shift;
    my $key = shift;
    my $value = shift;
    $self->{$key}=$value;
}

```

**EXAMPLE 14.5 (CONTINUED)**

```

5  sub FIRSTKEY {
      my $self = shift;
6      my $tmp = scalar keys %{$self};
7      return each %{$self};
    }
8  sub NEXTKEY {
      $self=shift;
      each %{$self};
    }
1;

-----

#!/usr/bin/perl
# File is mainfile
9  use House;
10 tie %home, "House";
    $home{"Price"} = 55000; # Assign and Store the data
    $home{"Rooms"} = 11;
    # Fetch the data
    print "The number of rooms in the house: $home{Rooms}\n";
    print "The price of the house is: $home{Price}\n";
11 foreach $key (keys(%home)){
12     print "Key is $key\n";
    }
13 while( ($key, $value) = each(%home)){
        # Calls to FIRSTKEY and NEXTKEY
14     print "Key=$key, Value=$value\n";
    }
15 untie(%home);

```

```

(Output)
I'm the constructor in package House
The number of rooms in the house: 11
The price of the house is: 55000
Key is Rooms
Key is Price
Key=Rooms, Value=11
Key=Price, Value=55000

```

**EXPLANATION**

- 1 The package/class *House* is declared. The file is *House.pm*.
- 2 The constructor *TIEHASH* will tie a hash to an object.
- 3 The *FETCH* method is an access method that will retrieve a value from the hash object.
- 4 The *STORE* method is an access method that will assign a value to the hash object.

**EXPLANATION (CONTINUED)**

- 5 The *FIRSTKEY* method is called automatically if the user program calls one of Perl's built-in hash functions: *keys*, *value*, or *each*.
- 6 By calling *keys* in a scalar context, Perl resets the internal state of the hash in order to guarantee that the next time *each* is called it will be given the first key.
- 7 The *each* function returns the first key/value pair.
- 8 The *NEXTKEY* method knows what the previous key was (*PREVKEY*) and starts on the next one as the hash is being iterated through a loop in the user program.
- 9 This is the user/driver program. The *House* module is loaded into the program.
- 10 The *tie* function calls the *TIEHASH* constructor in *House.pm*. The hash *%home* is tied to an object in the *House* class.
- 11 The *while* loop is used to iterate through the hash with the *keys* function. The first time in the loop, the *FIRSTKEY* method is automatically called.
- 12 The value for each key is printed. This value is returned from the access methods, *FIRSTKEY* and *NEXTKEY*.
- 13 The *while* loop is used to iterate through the hash with the *each* function. The first time in the loop, the *FIRSTKEY* method is automatically called.
- 14 Each key and value are printed. These values are returned from the access methods, *FIRSTKEY* and *NEXTKEY*.

**14.2 DBM Files**

The Perl distribution comes with a set of database management library files called DBM, short for database management. The concept of DBM files stems from the early days of UNIX and consists of a set of *C* library routines that allow random access to its records. DBM database files are stored as key/value pairs, an associative array that is mapped into a disk file. There are a number of flavors of DBM support, and they demonstrate the most obvious reasons for using tied hashes.

DBM files are binary. They can handle very large databases. The nice thing about storing data with DBM functions is that the data is persistent; that is, any program can access the file as long as the DBM functions are used. The disadvantage is that complex data structures, indexes, multiple tables, and so forth are not supported, and there is no reliable file locking and buffer flushing, making concurrent reading and updating risky.<sup>2</sup> File locking can be done with the Perl *flock* function, but the strategy for doing this correctly is beyond the scope of this book.<sup>3</sup>

So that you don't have to figure out which of the standard DBM packages to use, the *AnyDBM\_File.pm* module will get the appropriate package for your system from the stan-

2. Although Perl's *tie* function will probably replace the *dbmopen* function, for now we'll use this function because it's easier than *tie*.

3. For details on file locking, see Descartes, A., and Bunce, T., *Programming the Perl DBI*, O'Reilly & Associates, 2000, p. 35.

standard set in the standard Perl library. The *AnyDBM\_File* module is also useful if your program will run on multiple platforms. It will select the correct libraries for one of five different implementations:

**Table 14.1** DBM Implementations

odbm	“Old” DBM implementation found on UNIX systems and replaced by NDBM
ndbm	“New” DBM implementation found on UNIX systems
sdbm	Standard Perl DBM, provides cross-platform compatibility, but not good for large databases
gdbm	GNU DBM, a fast, portable DBM implementation; see <i>www.gnu.org</i>
bsd-db	Berkeley DB; found on BSD UNIX systems, most powerful of all the DBMs; see <i>www.sleepycat.com</i>

The following table comes from the documentation for *AnyDBM\_FILE* and lists some of the differences in the various DBM implementations. Type

`perldoc AnyDBM_File`

at your command line prompt.

	odbm	ndbm	sdbm	gdbm	bsd-db
	----	----	----	----	-----
Linkage comes w/ perl	yes	yes	yes	yes	yes
Src comes w/ perl	no	no	yes	no	no
Comes w/ many unix os	yes	yes[0]	no	no	no
Builds ok on !unix	?	?	yes	yes	?
Code Size	?	?	small	big	big
Database Size	?	?	small	big?	ok[1]
Speed	?	?	slow	ok	fast
FTPable	no	no	yes	yes	yes
Easy to build	N/A	N/A	yes	yes	ok[2]
Size limits	1k	4k	1k[3]	none	none
Byte-order independent	no	no	no	no	yes
Licensing restrictions	?	?	no	yes	no

### 14.2.1 Creating and Assigning Data to a DBM File

Before a database can be accessed, it must be opened by using the *dbmopen* function or the *tie* function. This binds the DBM file to an associative array (hash). Two files will be created: one file contains an index directory and has *.dir* as its suffix; the second file, ending in *.pag*, contains all the data. The files are not in a readable format. The *dbm* functions are used to access the data. These functions are invisible to the user.

Data is assigned to the hash, just as with any Perl hash, and an element removed with Perl's *delete* function. The DBM file can be closed with the *dbmclose* or the *untie* function.

## FORMAT

```
dbmopen(hash, dbfilename, mode);
tie(hash, Module, dbfilename, flags, mode);
```

## EXAMPLE 14.6

```
dbmopen(%myhash, "mydbmfile", 0666);
tie(%myhash, SDBM_File, "mydbmfile", O_RDWR|O_CREAT, 0640);
```

Perl's report writing mechanism is very useful for generating formatted data from one of the DBM files. The following examples not only illustrate how to create, add, delete, and close a DBM file, but also how create a Perl-style report.

## EXAMPLE 14.7

```
(The Script)
#!/usr/bin/perl
# Program name: makestates.pl
# This program creates the database using the dbm functions
1 use AnyDBM_File; # Let Perl pick the right dbm for your system
2 dbmopen(%states, "statedb", 0666a) || die;
# Create or open the database
3 TRY: {
4     print "Enter the abbreviation for your state. ";
5     chomp($abbrev=<STDIN>);
6     $abbrev = uc $abbrev; # Make sure abbreviation is uppercase
7     print "Enter the name of the state. ";
8     chomp($state=<STDIN>);
9     lc $state;
10    $states{$abbrev}="\u$state"; # Assign values to the database
11    print "Another entry? ";
12    $answer = <STDIN>;
13    redo TRY if $answer =~ /Y|y/;
14 }
15 dbmclose(%states); # Close the database
-----
```

**EXAMPLE 14.7 (CONTINUED)**

```
(The Command line)
10 $ ls
    makestates.pl statedb.dir statedb.pagb
-----
(Output)
4 Enter the abbreviation for your state. CA
5 Enter the name of the state. California
7 Another entry? y
    Enter the abbreviation for your state. me
    Enter the name of the state. Maine
    Another entry? y
    Enter the abbreviation for your state. NE
    Enter the name of the state. Nebraska
    Another entry? y
    Enter the abbreviation for your state. tx
    Enter the name of the state. Texas
    Another entry? n
```

- a. Permissions are ignored on Win32 systems.
- b. On some versions, only one file with a *.db* extension is created.

**EXPLANATION**

- 1 The *AnyDBM\_File* module selects the proper DBM libraries for your particular installation.
- 2 The *dbmopen* function binds a DBM file to a hash. In this case, the database file created is called *statedb* and the hash is called *%states*. If the database does not exist, a valid permission mode should be given. The octal mode given here is *0666*, read and write for all, on UNIX type systems.
- 3 The labeled block is entered.
- 4 The user is asked for input, the abbreviation of his state. This input will be used to fill the *%states* hash.
- 5 The user is asked to enter the name of his state.
- 6 The value *state* is assigned to the *%states* hash where the key is the abbreviation for the state. The *\u* escape sequence causes the first letter of the state to be uppercase. When this assignment is made the DBM file will be assigned the new value through a *tie* mechanism that takes place behind the scenes.
- 7 The user is asked to enter another entry into the DBM file.
- 8 If the user wants to add another entry to the DBM file, the program will go to the top of the block labeled *TRY* and start over.
- 9 The *dbmclose* function breaks the tie (by calling the *untie* function), binding the DBM file to the hash *%states*.
- 10 The listing displays the files that were created with the *dbmopen* function. The first file, *makestates.pl*, is the Perl script. The second file, *statedb.dir*, is the index file, and the last file, *statedb.pg*, is the file that contains the hash data.



**EXAMPLE 14.8 (CONTINUED)**

```
(Output)
  Abbreviation  State
  =====
  AR            Arizona
  CA            California
  ME            Maine
  NE            Nebraska
  TX            Texas
  WA            Washington
  =====
                Number of states: 6
```

**EXPLANATION**

- 1 The *AnyDBM\_File* module selects the proper DBM libraries for your particular installation.
- 2 The *dbmopen* function binds a DBM file to a hash. In this case, the database file opened is called *statedb* and the hash is called *%states*.
- 3 Now that the DBM file has been opened, the user can access the key/value pairs. The *sort* function combined with the *keys* function will sort out the keys in the *%states* hash.
- 4 The *foreach* loop iterates through the list of sorted keys.
- 5 Each time through the loop another value is retrieved from the *%states* hash, which is tied to the DBM file.
- 6 After adding one to the *\$total* variable (keeping track of how many entries are in the DBM file), the *write* function invokes the report templates to produce a formatted output.
- 7 The DBM file is closed; that is, the hash *%states* is disassociated from the DBM file.
- 8 This is the format template that will be used to put a header on the top of each page.
- 9 The period ends the template definition.
- 10 This is the format template for the body of each page printed to standard output. The picture line below is used to format the key/value pairs on the line below the picture.
- 11 This is the format template that will be invoked at the bottom of the report.

**14.2.3 Deleting Entries from a DBM File**

To empty the completed DBM file, you can use the *undef* function; for example, *undef %states* would clear all entries in the DBM file created in Example 14.8. Deleting a key/value pair is done simply by using the Perl built-in *delete* function on the appropriate key within the hash that was tied to the DBM file.

**EXAMPLE 14.9**

(The Script)

```
#!/bin/perl
# dbmopen is an older method of opening a dbm file but simpler
# than using tie and the SDBM_File module provided
# in the standard Perl library Program name: remstates.pl
1 use AnyDBM_File;
2 dbmopen(%states, "statedb", 0666) || die;
TRY: {
    print "Enter the abbreviation for the state to remove. ";
    chomp($abbrev=<STDIN>);
    $abbrev = uc $abbrev; # Make sure abbreviation is uppercase
3 delete $states{"$abbrev"};
    print "$abbrev removed.\n";
    print "Another entry? ";
    $answer = <STDIN>;
    redo TRY if $answer =~ /Y|y/; }
4 dbmclose(%states);
```

(Output)

```
5 $ remstates.pl
Enter the abbreviation for the state to remove. TX
TX removed.
Another entry? n
6 $ getstates.pl
Abbreviation      State
=====
AR                Arizona
CA                California
ME                Maine
NE                Nebraska
WA                Washington
=====
Number of states: 5
7 $ ls
getstates.pl      makestates.pl    rmstates.pl      statedb.dir
statedb.pag
```

**EXPLANATION**

- 1 The *AnyDBM\_File* module selects the proper DBM libraries for your particular installation.
- 2 The *dbmopen* function binds a DBM file to a hash. In this case, the database file opened is called *statedb* and the hash is called *%states*.
- 3 Now that the DBM file has been opened, the user can access the key/value pairs. The *delete* function will remove the value associated with the specified key in the *%states* hash tied to the DBM file.

**EXPLANATION (CONTINUED)**

- 4 The DBM file is closed; in other words, the hash *%states* is disassociated from the DBM file.
- 5 The Perl script *remstates.pl* is executed to remove Texas from the DBM file.
- 6 The Perl script *getstates.pl* is executed to display the data in the DBM file. Texas was removed.
- 7 The listing shows the files that were created to produce these examples. The last two are the DBM files created by *dbmopen*.

**EXAMPLE 14.10**

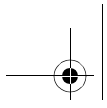
```

1 use Fcntl;
2 use SDBM_File;
3 tie(%address, 'SDBM_File', 'email.dbm', O_RDWR|O_CREAT, 0644)
  || die $!;
4 print "The package the hash is tied to: ", ref tied %address, "\n";

5 print "Enter the email address.\n";
  chomp($email=<STDIN>);
6 print "Enter the first name of the addressee.\n";
  chomp($firstname=<STDIN>);
  $firstname = lc $firstname;
  $firstname = ucfirst $firstname;
7 $address{"$email"}=$firstname;
8 while( ($email, $firstname)=each(%address)){
  print "$email, $firstname\n";
  }
9 untie %address;
```

**EXPLANATION**

- 1 The file control module is used to perform necessary tasks on the DBM files.
- 2 The *SDBM\_File* module is used. It is the DBM implementation that comes with standard Perl and works across platforms.
- 3 Instead of using the *dbmopen* function to create or access a DBM file, this example uses the *tie* function. The hash *%address* is tied to the package *SDBM\_File*. The DBM file is called *email.dbm*. If the database doesn't exist, the *O\_CREATE* flag will cause it to be created with read/write permissions (*O\_RDWR*).
- 4 The *tied* function returns true if it was successful and the *ref* function returns the name of the package where the hash is tied.
- 5 In this example, an e-mail address is used as the key in *%address* hash and the value associated with the key will be the first name of the user. Since the keys are always unique, this mechanism will prevent storing duplicate e-mail addresses. The user is asked for input.



**EXPLANATION (CONTINUED)**

- 6 The value for the key is requested from the user.
- 7 Here the database is assigned a new entry. The key/value pair is assigned and stored in the DBM file.
- 8 The *each* function will pull out both the key and value from the hash *%address*, which is tied to the DBM file. The contents of the DBM file are displayed.
- 9 The *untie* function disassociates the hash from the DBM file.

