Perl Programming, Part 2

Andrew Arensburger

March 3, 2000

Contents

Part II

1 Packages

A Perl package is similar to a C++ `namespace`: it defines a set of symbol tables into which to put variables, functions etc. This is useful for preventing different pieces of code in the same program from clobbering each other's variables.

```
package English;
$hello = "Hello";        # Sets $English::hello

package French;
$hello = "Bonjour";      # Sets $French::hello

package main;
print $hello;            # Wrong: $main::hello does not exist
print $French::hello;

package English;
print $hello;            # Prints $English::hello
```

When you use an identifier without a package name, such as `$foo` or `&myfunc`, Perl looks for them in the current package. To refer to a variable or function in a different package, prepend *packagename*`::` to the variable name, *e.g.*, `$French::hello`.

By default, variables and functions go in a package called `main`.

`package` *packagename* makes *packagename* the default package, *i.e.*, Perl will look for unqualified identifiers in package *packagename*. *packagename* will be the default package until the end of the enclosing scope, or until the end of the file if you do this at the top level.

You may switch packages multiple times, in different files if you like. You may do it anyplace you like, *e.g.*:

```
if ($language eq "english") {
        package English;
        print $hello;           # Print $English::hello
}                               # End of English scope
elsif ($language eq "french") {
        package French;
        print $hello;           # Print $French::hello
}                               # End of French scope
```

1.1 Nested Packages

Packages may also be nested, e.g.:

```
package Lang;
$hello = "Huh?";

package Lang::English;
$hello = "Hello";

package Lang::French;
$hello = "Bonjour";
print $hello;           # Prints $Lang::French::hello
```

However, this does not imply any sort of relationship between the outer and inner package. If your current package is `Lang::French` and you refer to `$hello`, but `$Lang::French::hello` is not defined, Perl will not look for `$Lang::hello`, nor will it look for `$main::hello`. Package nesting is solely for your convenience. If you want Perl to look for undefined identifiers in the parent package, you'll have to do that yourself. Fortunately, this is a common enough request that there are facilities for doing so.

1.2 BEGIN and END

Packages may contain `BEGIN` and `END` blocks. These act as initializers and finalizers for the package.[1]

The code in a `BEGIN` block runs before any other code, and that in an `END` block runs after everything else. In fact, `BEGIN` blocks are executed as soon as they are compiled, even before any following code has compiled:

```
package Moo;

print "Munch munch munch...\n";

BEGIN {
    print "Welcome to Moo.\n";
}
```

prints

```
Welcome to Moo.
Munch munch munch...
```

The code inside the `BEGIN` block will be executed first, during the compilation phase. In fact, it will be executed even if you're just checking the syntax of your Perl script. This allows `BEGIN` blocks to define functions that will be used later on, or pull in other files, and so forth.

You can have multiple `BEGIN` blocks in a package. They will be executed in the order in which they are seen.

[1] One might be tempted to say "constructors" and "destructors," but packages are not objects.

`END` blocks, on the other hand, are executed after everything else, after you call `exit` or `die`. You can have multiple `END` blocks in a package; they will be executed in the *reverse* order of that in which they were seen:

```
package Editor;
BEGIN { open THEFILE, "/some/file"; }
END   { close THEFILE; }

# Initialize lots of stuff.
$big_thing = &initialize();
END { &flush_changes($big_thing, THEFILE); }
```

Among other things, this allows a `BEGIN` block to abort compilation, and still have things closed cleanly.

Typically, however, you will only have at most one `BEGIN` and `END` block per package.

One caveat, though: since the intent of `BEGIN` and `END` blocks is to allow your code to modify the way the rest of the file is parsed, they will be executed even if you are only checking the syntax of your script. So be careful not to do anything drastic inside of them, like delete files or run daemons.

1.3 `AUTOLOAD`

If you call a function that hasn't been defined, Perl will look for the special function `AUTOLOAD` in the package where it expected to find the original function.

If you define an `AUTOLOAD` function, it will be called with the arguments to the original function. The variable `$AUTOLOAD` will be set to the name of the function that Perl was looking for.

The `AUTOLOAD` function should return whatever value the original function would have returned. It can define or load the function on the fly, or it can simply compute the return value and return it.

Once the `AUTOLOAD` function has been called, Perl assumes that all is well. If the function really cannot be called, then `AUTOLOAD` is responsible for either passing the buck to someone else, or `die`ing if necessary.

Thus, if you try to call `&MyPackage::somefunc(`*args*`)`, Perl will see if that function has been defined. If it hasn't, Perl will look for the function `MyPackage::AUTOLOAD`, and call it with the same arguments that the undefined function was called with.

Among other things, this is how you can "pass the buck" as mentioned earlier: if the package `Child` wants to inherit all of the functions in package `Parent` that it does not redefine, it can simply arrange to have its `AUTOLOAD` function call the appropriate function in package `Parent`.

2 Modules

Okay, so you've written a bunch of useful functions, you've put them in a separate module, and you'd like for people to use them. How do you do this?

2.1 `require`

`require "`*filename*`"` reads the contents of *filename*. This allows you to read in function and variable definitions from external files.

`@INC` is a list of directories. If you don't specify a full pathname to `require`, it will check each directory in `@INC` to see if it contains *filename*.

In addition, the hash `%INC` contains an entry for each file that has been included via `require`. This protects you from accidentally including a file twice.[2] This is like putting `#ifdef`...`#endif` around the contents of a `.h` file in C; Perl does this automatically for you.

A file read in with `require` should return a true value. If it returns a false value, this is interpreted as meaning that whatever the included file was meant to do cannot be done, and Perl will `die`. Usually, your file won't do anything more complex than define functions and set variables, so it's sufficient to add

```
1;
```

at the end of the file. If not, you can `return 0` at the top level of the package.

You can also use `require` *package*, in which case Perl will look for a file called *package*`.pm` (the `pm` extension stands for "Perl module").

Some operating systems have a problem with filenames containing `::`, so if you specify a nested package, its leading components will be interpreted as directory names by `require`. Thus, if you specify

```
require Lang::French::Canadian;
```

Perl will look for a file called `Lang/French/Canadian.pm`.

2.2 `use`

Typically, when you write a module, the entire module will be in one package, with the same name as the module file. If this is the case, you can use

`use` *module [list]*

instead of `require`.

```
use MyModule 1, 2, 3;
```

is almost exactly equivalent[3] to

```
BEGIN {
    require MyModule;
    MyModule::import("MyModule", 1, 2, 3);
}
```

[2] If you do want to include a file twice, you can `delete($INC{`*filename*`})`.

[3] Actually, it is exactly equivalent to

```
BEGIN {
    require MyModule;
    MyModule->import(1, 2, 3);
}
```

but we haven't seen `->` notation yet.

There are several things to note about the `use` statement: first of all, it implies a `BEGIN` block, which means that the `use` statement gets evaluated at compile-time, not at run-time. All of the functions that the module defines are compiled right at the beginning, instead of at some point during execution. If there are any syntax errors in your module, you can find out about them sooner rather than later. You also get the benefit of any function prototypes your module may contain.

`use` also calls the function `import` in the package you've just included.[4] This allows you to include "hooks" in your module, which will affect the way that the rest of the file is parsed.

For instance, consider the following:[5]

```
package Directories;

sub import {
    my $package = shift;

    # Add all arguments to the list of include directories
    unshift @INC, @_;
}

1;
```

You can save this as `Directories.pm` and use it as follows:

```
use Directories qw( /usr/local/test )
use LocalPackage;
```

Since `Directories::import` is run at compile-time, it modifies `@INC` just in time to be able to find the module `LocalPackage.pm`.

Typically, `import` functions are used to take functions that were defined in the imported package, and put them in the caller's namespace, so that the caller doesn't have to prepend `MyModule::` every time it calls one.

You may want to include a module at compile-time, but not run its `import` function (for example, you may not want your namespace modified), you can use

```
use MyModule();
```

which is equivalent to

```
BEGIN { require MyModule; }
```

[4] Note that `require` expects a filename, whereas the function call expects a package name. Hence, `use` will only work properly if the filename and package name are the same.

[5] This example is nothing more than a slightly simplified version of the `lib` module, from the standard Perl library.

2.3 no

Occasionally, you may want to turn off the effects of a module. In these cases, you may use

```
no MyModule;
```

This works the same way as `import MyModule`, but calls the `unimport` function, rather than `import`.

`no` is not widely used. Most of the time, it is used to temporarily disable the effects of `use strict`, which issues warnings about certain types of "unsafe" operations.

3 References

Consider the following code:

```
sub nroff { ... };
sub troff { ... };

if ($using_nroff) {
    &nroff("-ms", $filename);
} else {
    &troff("-ms", $filename);
}
```

The only difference between the if-clause and the else-clause is the name of the function being called. Aside from causing extra typing on the programmer's part, this hides what the code is trying to do: the point is that the file is being passed to a typesetter, not whether there is an if-clause.

If you were asked to add a third clause, *e.g.*, to handle *groff*, you might be tempted to be lazy and find a better way to do this.

3.1 Symbolic References

The first way to do this is by use of a *symbolic reference*: instead of a variable name, use a text expression that returns the name of the variable:

```
$expr
${expr}
```

For example:

```
$var = 3;
$ref = "var";
$$ref = 4;                      # Sets $var
${$ref} = 5;                    # Ditto
${$ref."two"} = 6;              # Sets $vartwo
```

```
@$ref = ('a','b','c');          # Sets @var
@refs = ("foo", "bar", "baz");
${$refs[0]} = "howdy";          # Sets $foo
${$refs[0]}[0] = "quux";        # Sets $foo[0]
```

You can use a symbolic reference anywhere you can use an ordinary variable. With symbolic references, our previous example becomes

```
sub nroff { ... };
sub troff { ... };
sub groff { ... };

   if ($using_nroff) { $typesetter = "nroff" }
elsif ($using_troff) { $typesetter = "troff" }
elsif ($using_groff) { $typesetter = "groff" }

&$typesetter("-ms", $filename);
```

The main problem with symbolic references is that they act upon the name of the variable. If a symbolic reference is used in a different lexical context than is intended, *e.g.*, if it is used in a different package, it won't do what you expect:

```
package Passwd;

sub expand_gcos {
    my $username = shift;
    my $gcosref  = shift;

    $$gcosref =~ s/&/$username/;
                # Modifies $Passwd::name, not
                # $main::name
}

package main;

$name = "andrew &urger";
&Passwd::expand_gcos("arensb", "name");
```

Since symbolic references are so error-prone, you may want to turn them off completely. You can do this with

```
use strict 'refs';
```

which will cause Perl to exit with a run-time error if you use a symbolic reference.

3.2 Hard References

The other type of reference is the *hard reference*. Each Perl object, be it a scalar, an array, a function or whatever, is represented internally by a data structure known as a *thingy*.[6]

A hard reference works on the same general principle as a symbolic reference, but instead of using the name of the variable it points to, it uses the address of the underlying thingy.

Hard references work a lot like pointers in C, except that they can only point to existing thingies, and you can't do pointer arithmetic on them (but unlike references in C++, you do need to dereference them).

Syntactically, hard references are used the same way as symbolic references. They are created using the "address of" operator, \ (backslash), similar to the `&` operator in C:

```
$ref    = \$myvar;
$$ref   = 3;                    # Sets $myvar
${$ref} = 4;                    # Ditto

@array       = (1, 2, 3);
$arrayyref   = \@array;
shift @{$arrayyref};            # shift @array;
$elementref  = \$array[1];
$$elementref = "two";           # @array now (1, "two", 3)

sub hi { print "hello" }
$funcref = \&hi;
&$funcref;                      # Calls &hi;
```

Taking the address of a variable that doesn't exist will cause that variable to instantly spring into existence. As usual, use the `-w` switch to catch possible typos.

3.3 `ref`

In the expression `@$arrayref`, the `$` indicates that `arrayref` is a scalar (hard references are, technically, scalars); the `@` says to use the array that `$arrayref` refers to.

Dereferencing a reference into a different type from the original thingy, *e.g.*, `%$arrayref` or `&$arrayref` will cause Perl to exit with a run-time error.

To find out what a reference refers to, use the `ref` function. `ref(`*reference*`)` returns the type of the thingy that *reference* refers to:

`REF`
`SCALAR`
`ARRAY`
`HASH`
`CODE` — A function.
`GLOB` — Usually used for filehandles.

[6] Larry Wall, Tom Christiansen and Randal L. Schwartz, "Programming Perl," 2nd ed., O'Reilly, p. 244. Just so you know I'm not making this up.

If the argument of `ref` is not a reference, it returns an empty string.

Also, if you use a hard reference in a string context (without dereferencing it), it will expand to a string containing the type of the reference, and the address of the thingy it points to:

```
$ref = \@array;
print "$ref\n";
```

prints `ARRAY(0x74e88)`.

3.4 Anonymous References

Since hard references only look at a variable's thingy, and never its name, it's even possible to have *anonymous references*, *i.e.*, references to thingies that have no variable associated with them.

3.4.1 References to Anonymous Arrays

Anonymous arrays are created using square brackets, e.g.,

```
$arrayref = [ 'a', 'b', 'c' ];
```

3.4.2 References to Anonymous Hashes

Anonymous hashes are created using curly brackets, e.g.

```
$hashref = { Jan => 31,
             Feb => 29,
             Mar => 31,
             Apr => 30,
           }
```

3.4.3 References to Anonymous Functions

It is even possible to create anonymous functions. Since curly braces are already taken for anonymous hashes, simply use a `sub` statement, but omit the function name:

```
$funcref = sub { print "Hello"; }
```

3.5 Complex Data Structures and `->`

Okay, so you're probably still wondering why you might want an anonymous anything in the first place. Well, consider the fact that references are, technically, scalars. This means that you can have arrays of them, or put them in hashes. And if you have an array of references to arrays, this means that you have, in effect, an array of arrays.

```
@array2D = (                    # Array of arrays
        [ 0, 1, 2 ],
        [ 'a', 'b', 'c', 'd', 'e' ],
        [ 31, [28, 29], 31, 30, 31, 30,
          31, 31, 30, 31, 30, 31
        ],
);
```

As this example illustrates, multi-dimensional arrays are more akin to nested lists in Lisp than to multi-dimensional arrays in C. The lists need not be all of the same length, and you can have sub-arrays in the middle of an array.

In the same way, you can build arrays of hashes, hashes of arrays, or any combination of arrays, hashes, etc. Let's look at an example:

```
$complex =
        [ [ "Unix",
            "operating system",
            { gurus => [ "Brian",
                         "Dennis",
                         "Ken"
                       ]
            }
          ],
          [ "Perl",
            "language",
            { author => "Larry",
              gurus  => [ "Larry",
                          "Tom",
                          "Randal"
                        ]
            }
          ]
        ];
```

Let's say we want to get at a deeply-nested part of this structure, say, `"Randal"`. This can be daunting, so let's do this step by step. First of all, the expression we want will involve the reference `$complex` in some way, so let's begin with that:

```
$complex
```

`$complex` is a reference to an anonymous array (as you can confirm with `print ref($complex)`). We want the second element (the one beginning with `[ "Perl"`, so let's dereference `$complex`

```
@{$complex}
```

and subscript the result:

```
${$complex}[1]
```

We now have the second element in `$complex`, which is itself a reference to an array. We want the third element of this array (the anonymous hash containing `author` and `gurus`). So let's dereference and subscript what we have so far:

```
${${$complex}[1]}[2]
```

Now we have a reference to an anonymous hash. We want the entry for `gurus`, so let's dereference and subscript this once again:

```
${${${$complex}[1]}[2]}{"gurus"}
```

Now we have a reference to yet another anonymous array (the one containing `"Larry"`, `"Tom"` and `"Randal"`. We want the third element of this array, so let's dereference and subscript one last time:

```
${${${${$complex}[1]}[2]}{"gurus"}}[2]
```

And voilà!

```
print ${${${${$complex}[1]}[2]}{"gurus"}}[2], "\n";
```

prints `Randal`, as expected.

As you can tell, this is not the most readable syntax ever invented. As a form of syntactic sugar, Perl supports the `->` operator:[7]

```
$ $arrayref  [0]
${$arrayref} [0]
  $arrayref->[0]

$ $hashref  {string}
${$hashref} {string}
  $hashref->{string}
```

Note that `->` notation implies that the result is a scalar (that is, it implies the first `$` in `${$arrayref}[0]`). If you want an array or a hash, you need to explicitly use

```
@{$arrayref}
```

or

```
%{$hashref}
```

Fortunately, in our case, we can use the arrow operator, so we can simplify the above monstrosity in several stages:

[7] Astute readers may note a remarkable similarity to C's `(*ptr).field` and `ptr->field`.

```
${${${${$complex} [1]} [2]} {"gurus"}} [2]
${${${  $complex->[1]} [2]} {"gurus"}} [2]
${${    $complex->[1]->[2]} {"gurus"}} [2]
${      $complex->[1]->[2]->{"gurus"}} [2]
        $complex->[1]->[2]->{"gurus"}->[2]
```

As a further form of syntactic sugar, if you have successive subscripts joined by an arrow, you can omit the arrow. Thus,

```
$complex->[1]->[2]->{"gurus"}->[2]
```

becomes

```
$complex->[1][2]{"gurus"}[2]
```

Obviously, this last expression is the most readable, and you should use it in your code. However, when you're debugging and aren't quite sure what is a reference to what, you can always fall back on the more explicit notation.

3.6 Garbage Collection

Each thingy maintains a *reference count*, which indicates how many references there are to it (from symbol tables, reference variables, etc.). When the last reference disappears, Perlautomatically frees up the thingy for you. You do not need to do anything special.

Variables initially have a reference count of 1 (from the symbol table that contains them).

3.7 Closures

I glossed over anonymous functions, above, but they deserve closer study.

References to functions are useful primarily as callbacks, *i.e.*, functions to be called when some interesting event occurs, such as a button being pushed, or as a hook to be called while parsing a complex data structure.

One important concern, if you think about it, is how an anonymous function looks up variables, particularly if it is called outside of the package in which it was defined. Fortunately, the answer boils down to "Perl behaves the way it looks as if it should." Consider the following code:

```
package Numbers;

$counter = 0;   # Gratuitous counter: how many times
                # has an anonymous function been called?

sub makefunc {
```

```
        my $divisor = shift;

        # Build a function on the fly, and return it
        return sub {
            my $number = shift(@_);

            $counter++;

            # Return true if $number is
            # evenly divisible by $divisor
            return ($number % $divisor == 0);
        };
    }

    package main;

    $is_even   = &Numbers::makefunc(2);
    $isdivby10 = &Numbers::makefunc(10);

    foreach $n (1, 2, 3, 4, 5)
    {
        print "$n is ",
            &$is_even($n) ? "even" : "odd",
            "\n";
    }
```

Here, `makefunc` is clearly in the `Numbers` package, and it does not pose any problems. However, the function that `Numbers::makefunc` defines is *defined* in `Numbers`, but *called* in `main`. When it uses `$number`, `@_`, `$divisor` and `$counter`, which variables, exactly, is it using?

The answer is that anonymous functions act as *closures* with respect to lexical variables ("global" and `my` variables). What this means is that anonymous functions carry with them a copy of what their environment looked like when they were created.

So, to answer the question above: `$number` is a `my` variable in the anonymous functions pointed to by `$is_even` and `$isdivby10`, so it is entirely private.

`@_` is determined when the function is called. It is set to `(1)`, `(2)`, etc. just the way it looks as if it should.

For `$counter`, we look at the code: the code that creates the anonymous functions is inside `Numbers::makefunc`, which itself is inside the package `Numbers`. Ordinarily, if a function used `$counter` in that context, it would be referring to `$Numbers::counter`. Therefore, that's what the anonymous functions do.

`$divisor` is a bit trickier, but once again, we look at the code: the code that creates the anonymous functions is inside `&Numbers::makefunc`, and `$divisor` is a `my` variable inside `makefunc`. Therefore, when `&$is_even` refers to `$divisor`, it uses a private copy of a variable whose initial value is that of `makefunc`'s

`$divisor` *at the time when* `&$is_even` *was created.*

You can verify, by making the anonymous functions modify `$number`, that their two copies are independent.

4 Interlude: A Short–and Probably Inaccurate–History of Programming

I've been throwing a lot of ideas at you out of the blue, and thought it might be enlightening to see where some of our current notions about programming come from. In particular, before we talk about Perl objects, I'd like to place object-oriented programming in some sort of historical context.

In the Bad Old Days, everyone used assembly language—whichever assembly language one's machine used. Since the only way to do flow control was through JMPs ("goto" statements), everyone used that. When Fortran came along, it inherited the `GOTO` statement, since that was what everyone used. Life was bad, but simple.

In the late 1960's, structured programming became the word of the day.[8] One of the tenets of structured programming was that code should be broken up into functional blocks, and control should enter each block at exactly one point, and exit at exactly one point. This was so that code could not jump into the middle of a while loop, or jump out to some random location, since this tends to result in so-called "spaghetti code."

With time, it has become clear that structured programming is the way to go, with one exception: while each block should only have one entry point, it is okay to break out of it in the middle. This is why C and Perl allow a function to return at any point, and why Perl has the `last`, `next` and `redo` constructs. Clearly,

```
sub count_elements {
    my @list = @_;              # List of elements
    my $i;
    my $num_elements = 0;       # Number of elements in list

    return 0 if @list == ();    # Trivial case

    for ($i = 0; $i <= $#list; $i++) {
        next if &is_invalid($list[$i]);
                                # Ignore invalid elements
        last if &is_end_of_list($list[$i]);
                                # Nothing useful after this
        $num_elements++;
    }
    return $num_elements;
}
```

[8] See Dikjstra, Edsger, "Goto Statement Considered Harmful," *CACM*, Mar. 1968.

is better than

```
sub count_elements {
    my @list = @_;              # List of elements
    my $i;
    my $num_elements;           # Return value
    my $seen_end = 0;           # Have we reached the end
                                # of the list?

    if (@list == ()) {
        $num_elements = 0;
    } else {
        # @list is not empty
        for ($i = 0; ($i <= $#list) && !$seen_end; $i++) {
            if (&is_invalid($list[$i])) {
                # Do nothing
            } else {
                # Check for end of list
                if (&is_end_of_list($list[$i])) {
                    $seen_end = 1;
                } else {
                    # Current element is valid
                    $num_elements++;
                }
            }
        }
    }
    return $num_elements;
}
```

which languages such as Pascal force you to write.

Once you've accepted the idea that several lines of related code should be grouped together into a single block, you can easily extend this to data as well. Consider the following declaration:

```
/* Sales information */
int     serial_no[1024];
double  price[1024];
char *  customer[1024];
```

What does this mean? Can this program handle 1024 sales, each of which consists of a serial number, price, and customer? Or can the program remember 1024 regular customers, as well as handle 1024 sales, each of which consists of a serial number and price?

Pascal's `record`s and C's `struct`s allow one to make this distinction. The first case can be written as

```
struct {
```

```
        int     serial_no;
        double  price;
        char *  customer;
    } sales[1024];
```

and the second as

```
    struct {
        int     serial_no;
        double  price;
    } sales[1024];
    char *  customer[1024];
```

And, of course, it is a simple step from grouping related variables together, to having certain functions associated with certain data structures and no others. For instance, the standard C library function `strlen()` returns the length of a string, and only a string. It makes no sense to call `strlen()` with an integer as its first argument. Yet C does nothing to prohibit this.

C++, on the other hand, allows the user to define *methods*. These are functions that are associated with a particular type of data structure, as if they were just another field in a `struct`.

The above shows a growing tendency towards *encapsulation*, the notion that a) things should be packaged into neat little packages, and b) the library user should be shielded from the implementation details of the library.

The last element required for *object-oriented programming* is *inheritance*.

[Grady Booch. Get reference from Jamie.]

Let's say that someone has defined a data structure that defines a file. It has all of the methods that you might want, such as `open`, `read`, `chmod`, etc.

However, you are interested in manipulating JPEG image files. Clearly, such a beast *is* a file, but with something added: you ought to be able to `open` and `read` an image file, just like any other file; but you should also be able to `display` it and get its dimensions.

Under the encapsulation model presented so far, there are no good solutions. The best one is perhaps that used by the X Athena widgets, where a `Widget` is a `struct` cleverly designed to make it possible for other `struct`s to inherit its attributes, but this demands a lot of discipline on the widget programmer's part.

An object-oriented language like C++ or Smalltalk, however, makes it easy to say, "an image file *is* a generic file, but with such-and-such features added or changed." This *isa* relationship is known as *inheritance*.

Should your code require it, your inherited type can also be a parent class: a progressive JPEG image file *isa* JPEG file *isa* image file *isa* file.

One further consequence of inheritance is the idea of *polymorphism*, also known as multiple inheritance, the notion that an object can have several *isa* relationships. For instance, a CD player *isa* stereo component, but from the point of view of the salesman, it also *isa* thing that you can sell.

If the language you are programming in supports polymorphism, different parts of the code will see your object in different ways, and will call different methods, depending on what they are interested in.

Polymorphism is usually problematic, because the different views of an object may conflict. For instance, you might define a class for displaying JPEG files on a graphical display. This class is simultaneously a `JPEGFile` and a `Window`. Both `JPEGFile` and `Window` have methods called `open`, which do entirely different things. When a program calls your class's `open` method, which one does it mean?

As we've seen, Perl supports encapsulation (through packages) and complex data structures (through anonymous references). It also supports inheritance and polymorphism, and we'll look at that next.

5 Objects

Each thingy has a label that can hold the name of a package. Normally, this label is empty. If it is set to the name of a package, however, that thingy is said to be an *object*. If the package happens to have functions that deal with objects, it is called a *class*.

The way to set the value of the label is with the `bless` function. `bless` takes as its arguments a reference to the object to be blessed (not the object itself), and the name of a package to bless it into.

```
package Sneeze;

$var = [ "achoo" ];
bless $var, Sneeze;
```

blesses `$var` as an object in the `Sneeze` package. If you omit the package name, the object is blessed into the current package.

You can re-bless an object into a new class, but if you do so, the previous blessing is forgotten.

If you have a reference to an object, and wish to know what class the object has been blessed into, simply use `ref` as for any other reference. It will return the name of the referenced thingy's class.

5.1 Methods

So far, Perl objects aren't terribly useful. In order to do any interesting object-oriented programming, you need methods, *i.e.*, functions that deal with objects. There's nothing particularly special about methods in Perl: they just expect to be called in certain ways, as we'll see.

Traditionally, there are two types of methods: *instance methods* and *class methods*. Instance methods operate on an object in a given class. Class methods are associated with a particular class, but not a particular instance of that class.

5.1.1 Class Methods

Constructors, which create new objects in a particular class, are examples of class methods. A class method in Perl is just a function; it simply expects a class name as its first argument.

```
package User;

# Create a new User
sub new {
    my $class = shift;          # Class name
    my $newthing = {};          # The new object

    bless $newthing, $class;    # Bless the object
    return $newthing;
}

# Look up a User by username
sub lookup {
    my $class = shift;          # Class name
    my $name  = shift;          # Name to look for
    my $newuser  = {};          # The new object

    # Look up the User
    my ($uname, $passwd, $uid, $gid) =
        getpwnam($name);
    $newuser->{"uname"} = $uname;
    $newuser->{"uid"}   = $uid;
    $newuser->{"gid"}   = $gid;

    bless $newuser, $class;     # Bless the object
    return $newuser;
}
```

There are two ways to call a class method:

```
$user1  = new User;
$arensb = lookup User "arensb";
$bin    = lookup User ("bin");

$user2  = User->new;
$root   = User->lookup("root");
```

The first type of call, without the arrow, is called the *indirect object* form, since it is similar to English sentences like "give Jack the ball." Note that there is no comma between the class name and the method arguments.

The second form, with the arrow, is called the *object-oriented* form. In both cases, the method will be passed `"User"` as its first argument. There is nothing special about the class name; it's just a string. Use whichever one seems clearer.

There is no special syntax for constructors in Perl. A constructor is any function that returns a new object. The name `new` is a convention, nothing more. In the example above, both `new` and `loookup` are constructors.

Astute readers may have noticed that in both new and lookup, above, I used the two-argument form of bless, and blessed the object into whatever class the function was given, rather than specifically blessing them into class User. There are good reasons for this, having to do with inheritance, which we'll talk about shortly.

The User class, above, is fairly typical in that it is implemented as an anonymous hash. This allows you to define named fields, which behave very similarly to struct or class members in C++.

5.1.2 Instance Methods

Instance methods are methods that deal with a specific instance of a class. Again, there is nothing remarkable about an instance method in Perl. It just takes an object reference as its first argument:

```
package User;

sub get_fullname {
    my $self = shift;           # The object
    my $fullname;               # Return value

    $fullname = $self->{"gcos"};    # Get the GCOS field
    $fullname =~ s/&/\u$self->{uname}/;
                                # Replace & with username
    return $fullname;
}

sub add_to_groups {
    my $self   = shift;         # The object
    my @groups = shift;         # Groups to add user to

    push @{$self->{"groups"}}, @group;
}
```

Again, the name $self is just a convention, nothing more. If you want to call the local reference to the object $this or $theobject, go right ahead.

Instance methods are invoked the same way as class methods, except that instead of the class name, you pass it a reference to a particular object:

```
$user = new User;
get_fullname $user;
add_to_group $user "operator", "bin";

$arensb = User->lookup("arensb");
$arensb->get_fullname;
$arensb->add_to_group("wheel");
```

Again, use which ever form seems clearer.

5.1.3 Dual-Nature Methods

Since the difference between a class method, an instance method, and an ordinary function lies solely in the type of its first argument, a single function can fulfill all three roles. All it needs to do is find out whether it has an argument, and if so, whether that argument is a reference (in which case it's an instance method) or a string (in which case it's a class method).

For instance, you might make `new` a dual-nature method:

```
$user1 = new User;       # Create a new User from scratch
$arnie = new $arensb;    # Make $arnie a clone of $arensb
```

However, this might make your code confusing, so don't do this just for the thrill of it.

5.2 Inheritance

If you call `Package::function`, and `function` isn't defined in `Package`, Perl will look for an array `@Package::ISA`. `@ISA` is a list of package names that the package inherits from. If Perl can't find a function in a given package, it will look for it in all of the packages listed in `@ISA`, in the order listed. Thus, if you have

```
package Child;
@ISA = qw( Mother Father );

package main;
Child->say("hello");
```

Perl will first look for `Mother::say`; if it doesn't find it, it will look in the packages named in `@Mother::ISA`, and so forth. If Perl still hasn't found `say`, it will then look for `Father::say`, then in the packages named in `@Father::ISA`, and so on.

If it still hasn't found the function, Perl will then start looking for an `AUTOLOAD` function in all of the parent classes listed above.

If it still can't find anything, Perl will then look in the predefined package `UNIVERSAL`, which is a sort of last-ditch class that all classes implicitly inherit from. If even that doesn't work, Perl gives up and exits with an exception.

All of this, by the way, applies to all packages, not just those that happen to be used as classes. There's really nothing special about a class; it's just a package that happens to contain methods.

I mentioned above that polymorphism (in this case, having more than one entry in `@ISA`) was a bad thing, since one method could shadow another. If you know which class's method to use, you can call it explicitly:

```
$billy = new Child;
Father::say($billy, "Hello");
```

The problem with this is that it does not do any inheritance: if `Father::say` isn't defined, Perl won't look through `@Father::ISA`. If you want to tell Perl to start looking in class `Father`, but to look through `@Father::ISA` if necessary, use

```
$billy->Father::say("Hello");
```

Note that if Perl can't find `say` in `Father` or its parent classes, it will not go back and start looking in `Mother`. But then, presumably that's what you intended anyway.

A method may also use the `SUPER` pseudo-class, to invoke methods in its parent classes, without having to explicitly name them. This is useful for classes that want to do everything their parent does, plus a bit more:

```
package FilledPolygon;
@ISA = qw( Polygon );

sub draw {
    my $self = shift;

    $self->SUPER::draw;     # Look for parent's draw()
    &fillme;
}
```

Because a class may be subclassed, it is best if a method does not assume that it knows how it has been called. For instance,

```
package Parent;

sub new {
    my $newobject = {};

    bless $newobject;       # Bless into current package
    return $newobject;
}

package Child;
@ISA = qw( Parent );

package main;

$thing = new Child;
```

Here, the class `Child` does not have a `new` method, so it inherits one from `Parent`. The method `Parent::new`, however, uses the single-argument form of `bless`, which blesses the object in to the current package, in this case `Parent`.

If the main program then tries to call one of `$thing`'s methods, Perl will not look for them in the `Child` package. Hence, it is best to get the class name from the first argument, and to bless the new object into that class.

5.3 Destructors

Once the last reference to an object disappears, that object is destroyed. If its class has defined a method called `DESTROY`, it will be called, with the soon-to-be-demised object as an argument.

Destructors are not used very frequently, since Perl takes care of freeing memory for you. They're there as a hook, to allow you to clean up before the object disappears, *e.g.*, close any files that it may have opened.

Note that a class is responsible for calling its parents' destructors; Perl will not do this automatically. You may find `SUPER::DESTROY` useful, but if the class has multiple parents, this will only call the first one's destructor.

5.4 Privacy

Perl does not enforce privacy the way that C++ does. By convention, methods that should only be called from inside the class begin with an underscore (_) and are not listed in the user documentation.

If you choose to call a private function, however, Perl will let you, since presumably you know what you're doing.

Likewise, if you define a class that is not intended to be subclassed, the way to prevent people from subclassing it is by saying so in the documentation.

6 Odds and Ends

6.1 `eval`

The `eval` function takes one of two forms:

`eval` *string*
`eval` *block*

`eval` parses and evaluates its argument as a little Perl program. This is handy if you don't know precisely what you want to do ahead of time, or for executing user-supplied expressions.

The return value of `eval` is the value of the last expression inside it.

Here's a simple Perl shell:

```
sub AUTOLOAD { return system($AUTOLOAD, @_); }

while (<>)
{
    eval $_;
}
```

If at all possible (*i.e.*, if you know in advance what the code looks like), use the block form of `eval`. That way, Perl will parse and compile the block at compile-time, so if the block contains any syntax errors, you'll find out about them at compile-time instead of at run-time.

If the code inside of an `eval` exits unexpectedly, *i.e.* through `die`, the `eval` will set `$@` to the text of `die`'s error message. If the block exits normally, `$@` will be set to the empty string.

If you want exception-handling in Perl, this is the way to do it.

6.2 Typeglobs

The typeglob operator `*` isn't used very much these days, since references do most of what it does, and better.

`*foo` refers to `all` variables named `foo`: `$foo`, `@foo`, `%foo` and the filehandle `foo`.

```
*foo = *bar;
```

aliases everything named `foo` to the corresponding variable called `bar`. The standard exporting mechanism, whereby functions defined in a package are exported to another, so that they can be called without specifying a package name, is implemented this way.

If you don't want to be quite so universal, then

```
*foo = \$bar;
```

makes `$foo` an alias for `$bar`, but doesn't affect `@foo` or `%foo`.[9]

The other main use for typeglobs is for making references to filehandles:

```
open(MYFILE, ">/my/file");
$fh = \*MYFILE;
print $fh "Hello, world\n";
close MYFILE;
```

6.3 Tied Variables

Perl allows you to specify functions for manipulating variables. For instance, database files "pretend to be" hashes for convenience. You can do the same thing for other types.

Tied variables are not aliases: the functions used in tying allow you to control the low-level access to a variable. You can define a hash that behaves like a scalar, or a "virtual" variable that doesn't really exist at all, but is defined in terms of what happens when you use it.

Note that tying variables is very similar to overloading operators in C++, and should be avoided for the same reasons: you are, in effect, creating a magic variable, one that looks ordinary but might be doing all sorts of things in the

[9] The other way to do this is by manipulating the package's symbol table directly:

```
$MyPackage::{"foo"} = \$bar;
```

background. In general, it's best to give the programmer some hint that complicated things might be going on, so it's better if you define a class to do what you want.

To create a tied variable, use the `tie` function:

> `tie` *var*`,` *class [*`,` *list]*

and provide the necessary interface functions (see below).

`tie` returns a reference to the object underlying the tied variable. You can later pass this to `untie`. If you didn't keep a reference, you can get the underlying object using

> `tied` *var*

6.3.1 Tied Scalars

To tie a scalar, you need to define:

`TIESCALAR` *class*`,`*list* This performs the initial tying: `TIESCALAR` is passed the class name and the argument list from `tie`, and creates (and returns) the underlying object.

`FETCH` *this* `FETCH` is called whenever the tied variable is read. It is passed a reference to the underlying object. It should return the tied variable's value.

`STORE` *this*`,` *value* `STORE` is called whenever the code wants to set the value of the tied variable. It is given the underlying object, as well as the value that is being assigned.

Since assignment to a variable returns the value that was just assigned, your `STORE` function should do this as well.

`DESTROY` *this* As with any object, you can have a destructor for your tied variable. The `DESTROY` function is optional.

Here's a way to allow you to get or set the umask through a scalar variable:

```
package Umask;

sub TIESCALAR {
    my $class = shift;
    my $self = {};      # Dummy placeholder

    bless $self, $class;
    return $self;
}

sub FETCH {
    my $self = shift;
```

```
    return umask;
}

sub STORE {
    my $self = shift;
    my $value = shift;

    umask $value;
}

package main;

tie $umask, Umask;
```

6.3.2 Tied Arrays

To tie an array, you need to define:

- `TIEARRAY` *class*, *list* — `TIEARRAY` performs the initial tying. It works just like `TIESCALAR`.
- `FETCH` *this*, *index* — `FETCH` is called any time the code tries to get the value of an element in the tied array. `FETCH` is passed a reference to the underlying object, and the numeric index of the element the code is trying to read.
- `STORE` *this*, *index*, *value* — `STORE` is called when the code wants to store a value at a particular array index. It is passed a reference to the underlying object, the index at which to store the new value, and the new value itself.
- `DESTROY` *this* — Again, the destructor is optional.

Note that tied arrays are currently not particularly well supported. There is no support for `$#array` constructs, or for the standard deque functions `push`, `pop`, `shift` and `unshift`.

Here is a simple class that allows you to pretend that you have an array with every user's username in it. Writing to the array is left as an exercise to the reader.

```
package Uname;

sub TIEARRAY {
    my $class = shift;
    my @rest = @_;
    my $self = {};

    bless $self, $class;
}

sub FETCH {
```

```
    my $self = shift;
    my $index = shift;
    my $uname;

    ($uname) = getpwuid($index);
    return $uname;
}

sub STORE {
    my $self  = shift;
    my $index = shift;
    my $value = shift;

    die("Can't write to read-only array");
}

package main;

tie @users, Uname;
print $users[2072], "\n";
```

6.3.3 Tied Hashes

In earlier versions of Perl, database access was provided through `dbmopen` and related functions. In Perl 5, one uses tied hashes for this. Because of this history, hashes are the most complex and useful of the tied variables.

To tie a hash, you need to define:

`TIEHASH` *class*, *list* — `TIEHASH` performs the initial tying, and works just like `TIESCALAR` and `TIEARRAY`.

`FETCH` *this*, *key* — `FETCH` is used to retrieve a value from the hash. It is passed the underlying object and the key to look up.

`STORE` *this*, *key*, *value* — `STORE` is used to store a value in the hash. It is passed the underlying object, the key at which to store the value, and the value itself.

`DELETE` *this*, *key* — `DELETE` is used to delete a key-value pair from the hash. It is passed the underlying object, and the key to delete.

`CLEAR` *this* — `CLEAR` clears the hash, *e.g.*, if it was assigned an empty array.

`EXISTS` *this*, *key* — Recall that a key may have an undefined value. The `EXISTS` function should return a true value if the given key exists, whether or not its value is defined.

`FIRSTKEY` *this* — `FIRSTKEY` is used when the code starts iterating over the hash, *e.g.* via `each` or `keys`.

`NEXTKEY` *this*, *lastkey* Each time the code calls `each` or `keys` on the hash, the `NEXTKEY` method is called. Aside from the underlying object, it is also passed the last key that was returned. This can be helpful if the function needs to know the previous key in order to get the next one.

`DESTROY` *this* Again, the destructor is optional.

Usually, you'll want to use a tied hash to access a database file. There are several modules in the standard Perl library to do this, so consult the documentation for `AnyDBM_File.pm`, `DB_File.pm` and `NDBM_File.pm`. Generally, though, you'll use something of the form

```
use Fcntl;
use NDBM_File;

$db_filename = "/my/database";
tie %mydb, NDBM_file, $db_filename, O_RDWR|O_CREAT, 0644;
```

Just for completeness, here's a tied hash implementation. This one's a bit more complex than the others (and probably full of bugs), so a word of explanation is required.

Let's say you have an application MyApp, which reads a number of initialization files when it starts. It looks for these files in the directories specified by the `$MYAPPDIRS` environment variable. In addition, certain files are not located in the directories listed in `$MYAPPDIRS`, but you'd like to access them using the same mechanism as the other files. This class allows you to define aliases for these files.

```
package InitFiles;

# This package implements a list of directories containing
# files as a tied hash.

sub TIEHASH {
    my $class = shift;
    my @dirlist = @_;
    my $self = {
        dirs => [ @dirlist ],   # Directories to look in
        aliases => {},          # Aliases for files
    };

    bless $self, $class;
    return $self;
}

sub FETCH {
    my $self = shift;
    my $key = shift;
```

```
    # See if this filename is really an alias
    if (defined($self->{"aliases"}{$key}))
    {
        return $self->{"aliases"}{$key};
    }

    # If there's no alias, returns the full pathname
    # to the file.
    for (@{$self->{"dirs"}})
    {
        my $fullname = "$_/$key";

        if ( -f $fullname )
        {
            return $fullname;
        }
    }
}

sub STORE {
    my $self = shift;
    my $key = shift;
    my $value = shift;

    # Define an alias for this key
    $self->{"aliases"}{$key} = $value;
    return $value;
}

sub DELETE {
    my $self = shift;
    my $key = shift;

    # Undefine the alias for this filename
    delete $self->{"aliases"}{$key};
}

sub CLEAR {
    my $self = shift;

    # Delete all aliases and the search list
    $self->{"dirs"} = [];
    $self->{"aliases"} = {};
}
```

```
sub FIRSTKEY {
    my $self = shift;

    # Close any pending opendir()
    if (defined($self->{"dirhandle"}))
    {
        closedir *{$self->{"dirhandle"}};
        delete $self->{"dirhandle"};
    }

    # Make a list of directories to look in
    $self->{"toread"} = [@{$self->{"dirs"}}];

    # Open the first directory and read the first entry
    opendir DIR, $self->{"dirs"}[0];
    $self->{"dirhandle"} = \*DIR;

    return readdir DIR;
}

sub NEXTKEY {
    my $self = shift;

    # Read the next directory entry
    my $nextfile = readdir *{$self->{"dirhandle"}};

    return $nextfile if defined $nextfile;

    # The last readdir() failed. Close this directory.
    closedir *{$self->{"dirhandle"}};
    shift @{$self->{"toread"}};
    if (@{$self->{"toread"}} == ())
    {
        # Nothing left to read.
        delete $self->{"dirhandle"};
        return undef;
    }

    # Open the next directory
    opendir DIR, ${$self->{"toread"}}[0];
    $self->{"dirhandle"} = \*DIR;
    # And read the first entry in it
    return readdir DIR;
}

package main;
```

```
tie %files, InitFiles, $ENV{"MYAPPDIRS"}, "/usr/local/share/MyApp";
$files{"nonstandard"} = "/etc/MyApp.rc";

while (($key, $value) = each %files)
{
    print "[$key] -> [$value]\n";
}
```

6.4 Style

Even though the Perl motto is "There Is More Than One Way To Do It," that doesn't mean that all methods are equally good. Here are a few stylistic pointers that can help improve the readability and maintainability of your code.

• Use `next` and `last` to break out of loops and early on. Typically, you'll need to check your data to make sure it's valid. For instance, when you're reading lines from a file, you probably want to ignore blank lines and comments. You can do this with

```
while (<>)
{
    chomp;
    next if /^\s*$/;    # Ignore blank lines
    next if /^\s*#/;    # Ignore comments

    # Real processing goes here
}
```

If you test for such oddball conditions early on, then the rest of the loop can make assumptions about the data it's working with. In this case, the rest of the loop doesn't have to worry about comments, since it knows that's already been taken care of.

• `return` from functions early. This is an extension of the previous point. You can often test for trivial, base case, or abnormal inputs to a function fairly easily (*e.g.*, an empty argument list). If you handle these cases at the top of your function, the rest of the body can assume that it's working on valid, mainstream cases.

• Put functions at the end of your program, so that you can easily find the main body. Unlike C, Perl does not have a `main` function: anything that's not in a function is part of the main body. If you define functions at the top of the file, it may be hard to find the main body. If you want to call your functions as list operators (*i.e.*, without the `&`), simply declare them at the top.

• If you have any customization variables, set them as the very first thing in your file. I also like to give them names in all caps. This way, if someone wants to change, say, the default directory where your program looks for images, he'll find

```
$IMAGEDIR = "/usr/local/images";
```

in the first place he looks, at the top of your program.

- Choose wisely between prefix and postfix conditionals. The rule of thumb here is, if you're doing X, where X is some short statement, and the point is that you're doing X, rather than whether you're making a choice, use a postfix conditional or loop. For example:

```
next while /^\s*$/;
print if /BEGIN/../END/;
```

Otherwise, just use the traditional form.

- Choose wisely between `if` and `unless`. The implied "not" in `unless` can easily cause confusion. I never use the prefix form of `unless`, and only use the postfix form for short statements.

Also, to me, the difference between

```
next unless defined($key);
```

and

```
next if !defined($key);
```

is that the second form implies some sort of abnormal or exceptional occurrence, whereas the first merely means "skip over the empty entries."

- You Almost Always Want `my`, not `local`. `my` uses lexical scoping, just as in most programming languages you might be familiar with. This means that a `my` variable is not available outside of the braces where it was declared `my`.

`local`, on the other hand, uses dynamic scoping, and is something of a hack, in my opinion: `local $var` saves the *global* variable `$var` on a stack; then, when control exits the block where `$var` was delared `local`, the old value is popped again. This means that the following code

```
sub foo {
    local $value = "inner";

    &bar;
}

sub bar {
    print "value == $value";
}

$value = "outer";
&foo;
```

will print `"inner"`, not `"outer"` as you might expect.

7 Further Study

There's a lot more I'd like to talk about, but I have to stop somewhere, and here is as good a place as any. The following topics are interesting, and their associated manpage is adequate or better.

- Man page roadmap. The Perl documentation is too large to fit in one manpage. The perl(1) man page lists all of the other man pages.
- Command-line arguments. Some of them are quite useful, and allow you to write non-trivial scripts on the command line. See perlrun(1).
- PODs—Plain Old Documentation. This allows you to embed documentation inside of a Perl script. See perlpod(1).
- Tricks and techniques. The perlbot(1) man page lists a number of tips for dealing with objects. Other cute or cool tricks you can discover by reading others' code, and writing your own.
- Embedding and extending. You can extend Perl by writing libraries of functions that can be called from Perl scripts. TkPerl was built this way. On top of that, you can have your C program execute Perl statements. Very handy if you want to allow the user to customize your application. Nvi and Apache do this. Start with perlembed(1).

```
srand(@}=split $},"Pe thoe\nrslrtcaeu knrJa h");
print  splice  @},  rand @}  =>0<= 1  while  @};
```