

Default Variables

\$" is separator character for printing lists (originally " ")

\$_ is default scalar variable for lots of operators (also called **\$ARG**)

print

chomp

loop control variable

shift

string matching operators

@_ is argument passing list for subroutines

@ARGV is command line arguments

\$0 is program name

\$/ end of line code for **<>** operator

\$a and **\$b** are used in sort and should not be used elsewhere

\$<digits> subpattern from captured parentheses of last pattern match

\$+ or **\$LAST_PAREN_MATCH** last bracket of the last successful search pattern

^N text matched by the used group most-recently closed of the last successful search pattern

@+ or **@LAST_MATCH_END** all of the offsets of the ends of the last successful submatches in the currently active dynamic scope

\$* Set to a non-zero integer value to do multi-line matching within a string

\$. or **HANDLE->input_line_number(EXPR)** or

\$INPUT_LINE_NUMBER or **\$NR** Current line number for the last filehandle accessed

\$/ or **\$RS** or **IO::Handle->input_record_separator(EXPR)** or

\$INPUT_RECORD_SEPARATOR The input record separator, newline by default

\$| or **HANDLE->autoflush(EXPR)** or **\$OUTPUT_AUTOFLUSH** If set to nonzero, forces a flush right away and after every write or print on the currently selected output channel

\$, or **\$OFS** or **IO::Handle->output_field_separator EXPR** or **\$OUTPUT_FIELD_SEPARATOR** The output field separator for the print operator

\$ or **\$ORS** or **IO::Handle->output_record_separator EXPR** or **\$OUTPUT_RECORD_SEPARATOR** The output record separator for the print operator

\$" or **\$LIST_SEPARATOR** This is like **\$,** except that it applies to array and slice values interpolated into a double-quoted string. Default is a space.

\$; or **\$SUBSCRIPT_SEPARATOR** or **\$SUBSEP**

The subscript separator for multidimensional array emulation. If you refer to a hash element as

`$foo{$a,$b,$c}`

it really means

`$foo{join($;, $a, $b, $c)}`

\$\$ The output format for printed numbers

\$\$% or **HANDLE->format_page_number(EXPR)** or **\$FORMAT_PAGE_NUMBER** The current page number of the currently selected output channel

\$\$= or **HANDLE->format_lines_per_page(EXPR)** or **\$FORMAT_LINES_PER_PAGE** The current page length (printable lines) of the currently selected output channel

\$\$- or **HANDLE->format_lines_left(EXPR)** or **\$FORMAT_LINES_LEFT** The number of lines left on the page of the currently selected output channel

\$\$@- or **@LAST_MATCH_START** **\$\$-[0]** is the offset of the start of the last successful match. **\$\$-[n]** is the offset of the start of the substring matched by n-th subpattern

\$\$~ or **\$FORMAT_NAME** or **HANDLE->format_name(EXPR)** The name of the current report format for the currently selected output channel

`$^` or `$FORMAT_TOP_NAME` or `HANDLE-`

`>format_top_name(EXPR)` The name of the current top-of-page format for the currently selected output channel

`$_` or `$FORMAT_LINE_BREAK_CHARACTERS` or `IO::Handle-`

`>format_line_break_characters EXPR` The current set of characters after which a string may be broken to fill continuation fields

`$^L` or `$FORMAT_FORMFEED` or `IO::Handle->format_formfeed`

`EXPR` What formats output as a form feed

`$^A` or `$ACCUMULATOR` The current value of the `write()` accumulator for `format()` lines

`$?` or `$CHILD_ERROR` The status returned by the last pipe close, backtick (```) command, successful call to `wait()` or `waitpid()`, or from the `system()` operator

`{^ENCODING}` The object reference to the Encode object that is used to convert the source code to Unicode

`$_` or `$ERRNO` or `$OS_ERROR` If used numerically, yields the current value of the C `errno` variable or if used as a string, yields the corresponding system error string

`%!` Each element of `%!` has a true value only if `$_` is set to that value

`$^E` or `$EXTENDED_OS_ERROR` Error information specific to the current operating system

`$@` or `$EVAL_ERROR` The Perl syntax error message from the last `eval()` operator

`$$` or `$PID` or `$PROCESS_ID` The process number of the Perl running this script

`$<` or `$UID` or `$REAL_USER_ID` The real uid of this process

`$>` or `$EUID` or `$EFFECTIVE_USER_ID` The effective uid of this process

`$(or $GID or $REAL_GROUP_ID The real gid of this process`

`)` or `$EGID` or `$EFFECTIVE_GROUP_ID` The effective gid of this process

\$0 or \$PROGRAM_NAME Contains the name of the program being executed

\$[The index of the first element in an array, and of the first character in a substring

\$] The version + patchlevel / 1000 of the Perl interpreter

^C or \$COMPILING The current value of the flag associated with the -c switch

^D or \$DEBUGGING The current value of the debugging flags

^F or \$SYSTEM_FD_MAX The maximum system file descriptor, ordinarily 2

^H WARNING: This variable is strictly for internal use only

%^H WARNING: This variable is strictly for internal use only

^I or \$INPLACE_EDIT The current value of the inplace-edit extension

^M By default, running out of memory is an untrappable, fatal error. However, if suitably built, Perl can use the contents of **^M** as an emergency memory pool after die()ing

^O or \$OSNAME The name of the operating system under which this copy of Perl was built

{^OPEN} An internal variable used by PerlIO

^P or \$PERLDB The internal variable for debugging support

^R or \$LAST_REGEXP_CODE_RESULT The result of evaluation of the last successful (?{ code }) regular expression assertion

^S or \$EXCEPTIONS_BEING_CAUGHT Current state of the interpreter

^T or \$BASETIME The time at which the program began running

{^TAINT} Reflects if taint mode is on or off

{^UNICODE} Reflects certain Unicode settings of Perl

^V or \$PERL_VERSION The revision, version, and subversion of the Perl interpreter

^W or \$WARNING The current value of the warning switch

`$_{^WARNING_BITS}` The current set of warning checks enabled by the use warnings pragma

`$^X` or `$EXECUTABLE_NAME` The name used to execute the current copy of Perl, from C's argv[0].

`ARGV` The special filehandle that iterates over command-line filenames in `@ARGV`

usually accessed with `<>`

`$ARGV` contains the name of the current file when reading from `<>`

`@ARGV` The array `@ARGV` contains the command-line arguments intended for the script

`ARGVOUT` The special filehandle that points to the currently open output file when doing edit-in-place processing with `-i`

`@F` The array `@F` contains the fields of each line read in when autosplit mode is turned on

`@INC` The array `@INC` contains the list of places that the `do` `EXPR`, `require`, or `use` constructs look for their library files

`@_` Within a subroutine the array `@_` contains the parameters passed to that subroutine

`%INC` The hash `%INC` contains entries for each filename included via the `do`, `require`, or `use` operators

`%ENV` or `$ENV{expr}` The hash `%ENV` contains your current environment

`%SIG` or `$SIG{expr}` The hash `%SIG` contains signal handlers for signals

`$&` or `$MATCH` are last successful pattern match (performance penalty)

`$`` or `$PREMATCH` string before last successful match (performance penalty)

`$'` or `$POSTMATCH` string after last successful match (performance penalty)

Errors and Error Codes

The variables `$@`, `$!`, `^E`, and `$?` contain information about different types of error conditions that may appear during execution of a Perl program.

`$@` Perl interpreter

`$!` C library

`^E` operating system

`$?` an external program

Packages

Packages are like namespaces in C++

do not affect my() lexical variables

do affect local() variables

`$::var` references the main package and is the same as `$main::var`

`$Pack::var` references var in package Pack

' is also accepted for :: for backward compatability

`$x'y` is the same as `$x::y`

`$x::y::z` refers to package `x::y` which has no relation to package `x`

Only identifiers starting with letters (or underscore) are stored in a package's symbol table

All other symbols are kept in package main, including all punctuation variables, like `$_`.

In addition, when unqualified, the identifiers `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ARGVOUT`, `ENV`, `INC`, and `SIG` are forced to be in package main, even when used for other purposes than their built-in ones.

The symbol table for a package is stored in the hash of that name with two colons appended.

The main symbol table's name is thus `%main::`, or `%::` for short.

Assignment to a typeglob performs an aliasing operation, i.e.,

```
*dick = *richard;
```

causes variables, subroutines, formats, and file and directory handles accessible via the identifier `richard` also to be accessible via the identifier `dick`. If you want to alias only a particular variable or subroutine, assign a reference instead:

```
*dick = \$richard;
```

Which makes `$richard` and `$dick` the same variable, but leaves `@richard` and `@dick` as separate arrays.

Another use of symbol tables is for making "constant" scalars.

```
*PI = \3.14159265358979;
```

BEGIN, CHECK, INIT and END

Four specially named code blocks are executed at the beginning and at the end of a running Perl program. These are the BEGIN, CHECK, INIT, and END blocks.

A BEGIN code block is executed as soon as possible, that is, the moment it is completely defined, even before the rest of the containing file (or string) is parsed. You may have multiple BEGIN blocks within a file (or eval'ed string) -- they will execute in order of definition.

An END code block is executed as late as possible, that is, after perl has finished running the program and just before the interpreter is being exited, even if it is exiting as a result of a die() function. You may have multiple END blocks within a file--they will execute in reverse order of definition; that is: last in, first out (LIFO). END blocks are not executed when you run perl with the -c switch, or if compilation fails.

Note that END code blocks are not executed at the end of a string eval(): if any END code blocks are created in a string eval(), they will be executed just as any other END code block of that package in LIFO order just before the interpreter is being exited.

Inside an END code block, \$? contains the value that the program is going to pass to exit(). You can modify \$? to change the exit value of the program.

CHECK code blocks are run just after the initial Perl compile phase ends and before the run time begins, in LIFO order. CHECK code blocks are used in the Perl compiler suite to save the compiled state of the program.

INIT blocks are run just before the Perl runtime begins execution, in "first in, first out" (FIFO) order.

I/O

The statement

```
while (<STDIN>) {}
```

reads stdin a line at a time until the end of file binding \$₀ to the line read in

The statement

```
while (<>) { "deal with line" }
```

if there are no command line arguments

reads stdin a line at a time until the end of file binding \$₀ to the line read in

If there are command line arguments

effectively concatenates all of the files named by the command line arguments and reads them a line at a time until the end of all files binding \$₀ to the line read in

Data Structures - Description

The 5.0 release of Perl let us have complex data structures. You may now write something like this and all of a sudden, you'd have an array with three dimensions!

```
for $x (1 .. 10) {  
  for $y (1 .. 10) {  
    for $z (1 .. 10) {  
      $AoA[$x][$y][$z] = $x ** $y + $z;  
    }  
  }  
}
```

Alas, however simple this may appear, underneath it's a much more elaborate construct than meets the eye!

Let's look at each of these possible constructs in detail. There are separate sections on each of the following:

- * arrays of arrays
- * hashes of arrays
- * arrays of hashes
- * hashes of hashes
- * more elaborate constructs

But for now, let's look at general issues common to all these types of data structures.

References

Perl `@ARRAYs` and `%HASHes` are all internally one-dimensional. They can hold only scalar values (meaning a string, number, or a reference). They cannot directly contain other arrays or hashes, but instead contain references to other arrays or hashes.

You can't use a reference to an array or hash in quite the same way that you would a real array or hash. Just think of it as the difference between a structure and a pointer to a structure.

References are like pointers that know what they point to. This means that when you have something which looks to you like an access to a two-or-more-dimensional array and/or hash, what's really going on is that the base type is merely a one-dimensional entity that contains references to the next level. It's just that you can use it as though it were a two-dimensional one.

```
$array[7][12]          # array of arrays
$array[7]{string}     # array of hashes
$hash{string}[7]      # hash of arrays
$hash{string}{'another string'} # hash of hashes
```

Now, because the top level contains only references, if you try to print out your array in with a simple `print()` function, you'll get something that doesn't look very nice, like this:

```
@AoA = ( [2, 3], [4, 5, 7], [0] );
print $AoA[1][2];
7
print @AoA;
ARRAY (0x83c38) ARRAY (0x8b194) ARRAY (0x8b1d0)
```

If you want to get at the thing a reference is referring to, then you have to use either prefix typing indicators, like `$$blah`, `@$blah`, `@{$blah[$i]}`, or else postfix pointer arrows, like `$a->[3]`, `$h->{fred}`, or even `$ob->method()->[3]`.

Common Mistakes

The two most common mistakes made in constructing something like an array of arrays is either accidentally counting the number of elements or else taking a reference to the same memory location repeatedly.

```
for $i (1..10) {  
    @array = somefunc($i);  
    $AoA[$i] = @array;           # WRONG!  
}
```

this means

```
$counts[$i] = scalar @array;
```

Here's the case of taking a reference to the same memory location again and again:

```
for $i (1..10) {  
    @array = somefunc($i);  
    $AoA[$i] = \@array;         # WRONG!  
}
```

Here's the right way to do the preceding:

```
for $i (1..10) {  
    @array = somefunc($i);  
    $AoA[$i] = [ @array ];  
}
```

The square brackets make a reference to a new array with a copy of what's in @array at the time of the assignment.

So just remember always to use the array or hash constructors with [] or {}, and you'll be fine, although it's not always optimally efficient.

In summary:

```
$AoA[$i] = [ @array ];           # usually best  
$AoA[$i] = \@array;             # perilous; just  
how my() was that array?  
{ $AoA[$i] } = @array;         # way too tricky for  
most programmers
```

Caveat On Precedence

Speaking of things like `@{$AoA[$i]}`, the following are actually the same thing:

```
$aref->[2][2]          # clear  
$$aref[2][2]         # confusing
```

That's because Perl's precedence rules on its five prefix dereferencers (which look like someone swearing: `$ @ * % &`) make them bind more tightly than the postfix subscripting brackets or braces!

The seemingly equivalent construct in Perl, `$$aref[$i]` first does the deref of `$aref`, making it take `$aref` as a reference to an array, and then dereference that, and finally tell you the *i*'th value of the array pointed to by `$AoA`. If you wanted the C notion, you'd have to write `#{ $AoA[$i] }` to force the `$AoA[$i]` to get evaluated first before the leading `$` dereferencer.

Why You Should Always Use Strict

The best way to avoid getting confused is to start every program like this:

```
#!/usr/bin/perl -w
use strict;
```

This way, you'll be forced to declare all your variables with `my()` and also disallow accidental "symbolic dereferencing".

Therefore if you'd done this:

```
my $aref = [
  [ "fred", "barney", "pebbles", "bambam", "dino", ],
  [ "homer", "bart", "marge", "maggie", ],
  [ "george", "jane", "elroy", "judy", ],
];

print $aref[2][2];
```

The compiler would immediately flag that as an error at compile time, because you were accidentally accessing `@aref`, an undeclared variable, and it would thereby remind you to write instead:

```
print $aref->[2][2]
```

Debugging

Before version 5.002, the standard Perl debugger didn't do a very nice job of printing out complex data structures. With 5.002 or above, the debugger includes several new features, including command line editing as well as the x command to dump out complex data structures. For example, given the assignment to \$AoA above, here's the debugger output:

```
DB<1> x $AoA
$AoA = ARRAY(0x13b5a0)
  0  ARRAY(0x1f0a24)
    0  'fred'
    1  'barney'
    2  'pebbles'
    3  'bambam'
    4  'dino'
  1  ARRAY(0x13b558)
    0  'homer'
    1  'bart'
    2  'marge'
    3  'maggie'
  2  ARRAY(0x13b540)
    0  'george'
    1  'jane'
    2  'elroy'
    3  'judy'
```

Code Examples

here are short code examples illustrating access of various types of data structures.

Arrays Of Arrays

Declaration of an ARRAY OF ARRAYS

```
@AoA = (  
    [ "fred", "barney" ],  
    [ "george", "jane", "elroy" ],  
    [ "homer", "marge", "bart" ],  
);
```

Generation of an ARRAY OF ARRAYS

```
# reading from file  
while ( <> ) {  
    push @AoA, [ split ];  
}  
  
# calling a function  
for $i ( 1 .. 10 ) {  
    $AoA[$i] = [ somefunc($i) ];  
}  
  
# using temp vars  
for $i ( 1 .. 10 ) {  
    @tmp = somefunc($i);  
    $AoA[$i] = [ @tmp ];  
}
```

```
# add to an existing row  
push @{$AoA[0]}, "wilma", "betty";
```

Access and Printing of an ARRAY OF ARRAYS

```
# one element  
$AoA[0][0] = "Fred";  
  
# another element  
$AoA[1][1] =~ s/(\w)/\u$1/;
```

```
# print the whole thing with refs
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoA ) {
    print "\t [ @{$AoA[$i]} ],\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. ${ $AoA[$i] } ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}
```

HASHES OF ARRAYS

Declaration of a HASH OF ARRAYS

```
%HoA = (  
  flintstones => [ "fred", "barney" ],  
  jetsons     => [ "george", "jane", "elroy" ],  
  simpsons    => [ "homer", "marge", "bart" ],  
);
```

Generation of a HASH OF ARRAYS

```
# reading from file  
# flintstones: fred barney wilma dino  
while ( <> ) {  
  next unless s/^(.*?):\s*//;  
  $HoA{$1} = [ split ];  
}  
  
# reading from file; more temps  
# flintstones: fred barney wilma dino  
while ( $line = <> ) {  
  ($who, $rest) = split /\s*/, $line, 2;  
  @fields = split ' ', $rest;  
  $HoA{$who} = [ @fields ];  
}  
  
# calling a function that returns a list  
for $group ( "simpsons", "jetsons", "flintstones" ) {  
  $HoA{$group} = [ get_family($group) ];  
}  
  
# likewise, but using temps  
for $group ( "simpsons", "jetsons", "flintstones" ) {  
  @members = get_family($group);  
  $HoA{$group} = [ @members ];  
}  
  
# append new members to an existing family  
push @{ $HoA{"flintstones"} }, "wilma", "betty";
```

Access and Printing of a HASH OF ARRAYS

```
# one element
$HoA{flintstones}[0] = "Fred";

# another element
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing with indices
foreach $family ( keys %HoA ) {
    print "family: ";
    foreach $i ( 0 .. ${ $HoA{$family} } ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} }
keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing sorted by number of members and name
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}}
    || $a cmp $b
    } keys %HoA )
{
    print "$family: ",
        join(", ", sort @{ $HoA{$family} } ),
        "\n";
}
}
```

Arrays Of Hashes

Declaration of an ARRAY OF HASHES

```
@AoH=({Lead=>"fred", Friend=>"barney"},
      {Lead=>"george", Wife=>"jane", Son =>"elroy"},
      {Lead=>"homer", Wife=>"marge", Son=>"bart"}
    );
```

Generation of an ARRAY OF HASHES

```
# reading from file  format: LEAD=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @AoH, $rec;
}

# same with no temp
while ( <> ) {
    push @AoH, { split /\s+=/ };
}

# calling a function that returns a key/value pair list, like
# "lead", "fred", "daughter", "pebbles"
while ( %fields = getnextpairset() ) {
    push @AoH, { %fields };
}

# likewise, but using no temp vars
while (<>) {
    push @AoH, { parsepairs($_) };
}

# add key/value to an element
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";
```

Access and Printing of an ARRAY OF HASHES

```
# one element
$AoH[0]{lead} = "fred";

# another element
$AoH[1]{lead} =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoH ) {
    for $role ( keys %{ $AoH[$i] } ) {
        print "elt $i $role is $AoH[$i]{$role}\n";
    }
}
}
```

Hashes Of Hashes

Declaration of a HASH OF HASHES

```
%HoH=(flintstones=>{lead=>"fred", pal=>"barney"},
      jetsons=>{lead=>"george", wife=>"jane", "his
boy"=>"elroy"},
      simpsons=>{lead=>"homer", wife=>"marge",
kid=>"bart"},
);
```

Generation of a HASH OF HASHES

```
# reading from file
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }
}

# reading from file; more temps
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

# calling a function that returns a key,value hash
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

# likewise, but using temps
```

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
    %members = get_family($group);
    $HoH{$group} = { %members };
}

# append new members to an existing family
%new_folks = (
    wife => "wilma",
    pet  => "dino",
);

for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}
```

Access and Printing of a HASH OF HASHES

```
# one element
$HoH{flintstones}{wife} = "wilma";

# another element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing somewhat sorted
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
```

```

# print the whole thing sorted by number of members
foreach $family ( sort { keys %{$HoH{$b}} <=> keys
%{$HoH{$a}} } keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{$HoH{$family}} ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# establish a sort order (rank) for each role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_}
= ++$i }

# now print the whole thing sorted by number of
members
foreach $family ( sort { keys %{$HoH{$b}} <=> keys
%{$HoH{$a}} } keys %HoH ) {
    print "$family: { ";
    # and print these according to rank order
    for $role ( sort { $rank{$a} <=> $rank{$b} } keys
%{$HoH{$family}} ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

```

MORE ELABORATE RECORDS

Declaration of MORE ELABORATE RECORDS

Here's a sample showing how to create and use a record whose fields are of many different sorts:

```
$rec = {
    TEXT      => $string,
    SEQUENCE  => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE  => \&some_function,
    THISCODE  => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};

print $rec->{TEXT};

print $rec->{SEQUENCE}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# careful of extra block braces on fh ref
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");
```

AUTHOR

Tom Christiansen tchrist@perl.com

Objects

Objects are just hashes that have been "blessed" so they know their package

Perl Classes

There is no special class syntax in Perl, but a package may act as a class if it provides subroutines to act as methods. Such a package may also derive some of its methods from another class (package) by listing the other package name(s) in its global @ISA array (which must be a package global, not a lexical).

Perl Classes

A Perl object is a special reference that has been blessed so you may dereference it using not merely string or numeric subscripts as with simple arrays and hashes, but with named subroutine calls or methods.

Creating a Class

A class is created in a package with the same name

This specifies the file name as in regular modules

Then, that class (package) should provide one or more ways to generate objects.

Finally, it should provide mechanisms to allow users of its objects to indirectly manipulate these objects from a distance.

For example, let's make a simple Person class module. It gets stored in the file Person.pm.

Do not assume any formal relationship between modules based on their directory names. This is merely a grouping convenience, and has no effect on inheritance, variable accessibility, or anything else.

In order to manufacture objects, a class needs to have a *constructor method*. A constructor gives you back a brand-new object in that class. This magic is taken care of by the `bleed()` function, whose sole purpose is to enable its referent to be used as an object. Remember: being an object really means nothing more than that methods may now be called against it.

While a constructor may be named anything you'd like, most Perl programmers seem to like to call theirs `new()`. However, `new()` is not a reserved word, and a class is under no obligation to supply such.

Object Representation

By far the most common mechanism used in Perl to represent a Pascal record, a C struct, or a C++ class is an anonymous hash. That's because a hash has an arbitrary number of data fields, each conveniently accessed by an arbitrary name of your own devising.

If you were just doing a simple struct-like emulation, you would likely go about it something like this:

```
$rec = {  
    NAME    => "Jason",  
    AGE     => 23,  
    PEERS   => [ "Norbert", "Rhys", "Phineas"],  
};
```

And so you could get at `$rec->{NAME}` to find "Jason", or `@{ $rec->{PEERS} }` to get at "Norbert", "Rhys", and "Phineas".

An object should be considered an opaque cookie that you use *object methods* to access. Visually, methods look like you're dereffing a reference using a function name instead of brackets or braces.

Class Interface

Some languages provide a formal syntactic interface to a class's methods, but Perl does not. It relies on you to read the documentation of each class. If you try to call an undefined method on an object, Perl won't complain, but the program will trigger an exception while it's running.

Let's suppose you have a well-educated user of your `Person` class, someone who has read the docs that explain the prescribed interface. Here's how they might use the `Person` class:

```
use Person;

$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( "Norbert", "Rhys", "Phineas" );

push @All_Recs, $him; # save object in array for later

printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", $him->peers), "\n";

printf "Last rec's name is %s\n", $All_Recs[-1]->name;
```

As you can see, the user of the class doesn't know that the object has one particular implementation or another. The interface to the class and its objects is exclusively via methods, and that's all the user of the class should ever play with.

Constructors and Instance Methods

Here's how to implement the Person class using the standard hash-ref-as-an-object idiom. We'll make a class method called new() to act as the constructor, and three object methods called name(), age(), and peers() to get at per-object data hidden away in our anonymous hash.

```
package Person;
use strict;

## the object constructor (simplistic version) ##
sub new {
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless($self); # but see section on inheritance
    return $self;
}

## methods to access per-object data ##
## ##
## With args, they set the value. Without ##
## any, they only retrieve it/them. ##

sub name {
    my $self = shift;
    if (@_) { $self->{NAME} = shift }
    return $self->{NAME};
}

sub age {
    my $self = shift;
    if (@_) { $self->{AGE} = shift }
    return $self->{AGE};
}
```

```
sub peers {  
  my $self = shift;  
  if (@_) { @{$self->{PEERS}} = @_ }  
  return @{$self->{PEERS}};  
}
```

```
1; # so the require or use succeeds
```

We've created three methods to access an object's data, name(), age(), and peers(). These are all substantially similar. If called with an argument, they set the appropriate field; otherwise they return the value held by that field, meaning the value of that hash key.

Planning for the Future: Better Constructors

To ensure that this all works out smoothly for inheritance, you must use the double-argument form of `bless()`. The second argument is the class into which the referent will be blessed. By not assuming our own class as the default second argument and instead using the class passed into us, we make our constructor inheritable.

```
sub new {
    my $class = shift;
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless ($self, $class);
    return $self;
}
```

That's about all there is for constructors. These methods bring objects to life, returning neat little opaque bundles to the user to be used in subsequent method calls.

Destructors

Any per-object clean-up code is placed in the destructor, which must (in Perl) be called **DESTROY**.

The only situation where Perl's reference-based GC won't work is when there's a circularity in the data structure, such as:

```
$this->{WHATEVER} = $this;
```

Other Object Methods

There's no reason to limit methods to those that simply access data. Methods can do anything at all. Let's say we'd like object methods that do more than fetch or set one particular field.

```
sub exclaim {
    my $self = shift;
    return sprintf "Hi, I'm %s, age %d, working
with %s",
                $self->{NAME}, $self->{AGE}, join(", ",
@{$self->{PEERS}});
}
```

Or maybe even one like this:

```
sub happy_birthday {
    my $self = shift;
    return ++$self->{AGE};
}
```

But since these methods are all executing in the class itself, this may not be critical. There are tradeoffs to be made. Using direct hash access is faster (about an order of magnitude faster, in fact), and it's more convenient when you want to interpolate in strings. But using methods (the external interface) internally shields not just the users of your class but even you yourself from changes in your data representation.

Class Data

You *could* make it a global variable called `$Person::Census`. But about only reason you'd do that would be if you *wanted* people to be able to get at your class data directly.

But more often than not, you just want to make your class data a file-scoped lexical. To do so, simply put this at the top of the file:

```
my $Census = 0;
```

Even though the scope of a `my()` normally expires when the block in which it was declared is done (in this case the whole file being required or used), Perl's deep binding of lexical variables guarantees that the variable will not be deallocated, remaining accessible to functions declared within that scope. This doesn't work with global variables given temporary values via `local()`, though.

Irrespective of whether you leave `$Census` a package global or make it instead a file-scoped lexical, you should make these changes to your `Person::new()` constructor:

```
sub new {
    my $class = shift;
    my $self = {};
    $Census++;
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless ($self, $class);
    return $self;
}

sub population {
    return $Census;
}
```

Now that we've done this, we certainly do need a destructor so that when `Person` is destroyed, the `$Census` goes down. Here's how this could be done:

```
sub DESTROY { --$Census }
```

Accessing Class Data

It turns out that this is not really a good way to go about handling class data. A good scalable rule is that *you must never reference class data directly from an object method*. Otherwise you aren't building a scalable, inheritable class. The object must be the rendezvous point for all operations, especially from an object method. The globals (class data) would in some sense be in the "wrong" package in your derived classes. In Perl, methods execute in the context of the class they were defined in, *not* that of the object that triggered them. Therefore, namespace visibility of package globals in methods is unrelated to inheritance.

If some other class "borrowed" (well, inherited) the DESTROY method as it was defined above. When those objects are destroyed, the original \$Census variable will be altered, not the one in the new class's package namespace. Perhaps this is what you want, but probably it isn't.

Here's how to fix this. We'll store a reference to the data in the value accessed by the hash key "_CENSUS".

```
sub new {
    my $class = shift;
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    # "private" data
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub population {
    my $self = shift;
    if (ref $self) {
        return ${ $self->{"_CENSUS"} };
    } else {
        return $Census;
    }
}

sub DESTROY {
    my $self = shift;
    -- ${ $self->{"_CENSUS"} };
}
```

Class Destructors

The object destructor handles the death of each distinct object. But sometimes you want a bit of cleanup when the entire class is shut down, which currently only happens when the program exits. To make such a *class destructor*, create a function in that class's package named END. This works just like the END function in traditional modules, meaning that it gets called whenever your program exits unless it execs or dies of an uncaught signal. For example,

```
sub END {  
    if ($Debugging) {  
        print "All persons are going away now.\n";  
    }  
}
```

When the program exits, all the class destructors (END functions) are be called in the opposite order that they were loaded in (LIFO order).

Inheritance

Perl has no special syntax for specifying the class (or classes) to inherit from. Each package can have a variable called `@ISA`, which governs (method) inheritance. If you try to call a method on an object or class, and that method is not found in that object's package, Perl then looks to `@ISA` for other packages to go looking through in search of the missing method.

Like the special per-package variables recognized by `Exporter` (such as `@EXPORT`, `@EXPORT_OK`, `@EXPORT_FAIL`, `%EXPORT_TAGS`, and `$VERSION`), the `@ISA` array *must* be a package-scoped global and not a file-scoped lexical created via `my()`. Most classes have just one item in their `@ISA` array. In this case, we have what's called "single inheritance", or SI for short.

Consider this class:

```
package Employee;
use Person;
@ISA = ("Person");
1;
```

All it's doing so far is loading in another class and stating that this one will inherit methods from that other class if need be. We have given it none of its own methods. We rely upon an `Employee` to behave just like a `Person`.

Setting up an empty class like this is called the "empty subclass test"; that is, making a derived class that does nothing but inherit from a base class. If the original base class has been designed properly, then the new derived class can be used as a drop-in replacement for the old one.

By proper design, we mean always using the two-argument form of `bless()`, avoiding direct access of global data, and not exporting anything.

A method is just a function that expects as its first argument a class name (package) or object (blessed reference): method calls get an extra argument. Also, function calls don't do inheritance, but methods do.

Method Call	Resulting Function Call
-----	-----
Person->new()	Person::new("Person")
Employee->new()	Person::new("Employee")

So don't use function calls when you mean to call a method.

If an employee is just a Person, that's not all too very interesting. So let's add some other methods. We'll give our employee data fields to access their salary, their employee ID, and their start date.

If you're getting a little tired of creating all these nearly identical methods just to get at the object's data, do not despair. Later, we'll describe several different convenience mechanisms for shortening this up. Meanwhile, here's the straight-forward way:

```

sub salary {
    my $self = shift;
    if (@_) { $self->{SALARY} = shift }
    return $self->{SALARY};
}

sub id_number {
    my $self = shift;
    if (@_) { $self->{ID} = shift }
    return $self->{ID};
}

sub start_date {
    my $self = shift;
    if (@_) { $self->{START_DATE} = shift }
    return $self->{START_DATE};
}

```

Overridden Methods

What happens when both a derived class and its base class have the same method defined? Well, then you get the derived class's version of that method.

Every now and then you'll want to have a method call trigger both its derived class (also known as "subclass") version as well as its base class (also known as "superclass") version. In practice, constructors and destructors are likely to want to do this, and it probably also makes sense for a debug() method.

```
$self->Person::debug($Debugging);  
$self->SUPER::debug($Debugging);
```

This way it starts looking in my class's @ISA. This only makes sense from within a method call, though. Don't try to access anything in SUPER:: from anywhere else, because it doesn't exist outside an overridden method call. Note that SUPER refers to the superclass of the current package, not to the superclass of \$self.

Multiple Inheritance

All this means is that rather than having just one parent class who in turn might itself have a parent class, etc., that you can directly inherit from two or more parents.

The way it works is simple: just put more than one package name in your `@ISA` array. When it comes time for Perl to go finding methods for your object, it looks at each of these packages in order. It's actually a fully recursive, depth-first order.

UNIVERSAL: The Root of All Objects

Perl tacitly and irrevocably assumes that there's an extra element at the end of @ISA: the class UNIVERSAL. In version 5.003, there were no predefined methods there, but you could put whatever you felt like into it.

However, as of version 5.004, UNIVERSAL has some methods in it already. Predefined methods include isa(), can(), and VERSION().

isa() tells you whether an object or class "is" another one without having to traverse the hierarchy yourself:

```
$has_io = $fd->isa("IO::Handle");  
$itza_handle = IO::Socket->isa("IO::Handle");
```

The can() method, called against that object or class, reports back whether its string argument is a callable method name in that class. In fact, it gives you back a function reference to that method:

```
$his_print_method = $obj->can('as_string');
```

Finally, the VERSION method checks whether the class (or the object's class) has a package global called \$VERSION that's high enough.

Alternate Object Representations

Nothing requires objects to be implemented as hash references. An object can be any sort of reference so long as its referent has been suitably blessed. That means scalar, array, and code references are also fair game.

A scalar would work if the object has only one datum to hold. An array would work for most cases, but makes inheritance a bit dodgy because you have to invent new indices for the derived classes.

Closures as Objects

Using a code reference to represent an object offers some fascinating possibilities. We can create a new anonymous function (closure) who alone in all the world can see the object's data. This is because we put the data into an anonymous hash that's lexically visible only to the closure we create, bless, and return as the object. This object's methods turn around and call the closure as a regular subroutine call, passing it the field we want to affect. (Yes, the double-function call is slow, but if you wanted fast, you wouldn't be using objects at all, eh? :-)

Use would be similar to before:

```
use Person;
$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( [ "Norbert", "Rhys", "Phineas" ] );
printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", @{$him->peers}), "\n";
```

but the implementation would be radically different:

```
package Person;
sub new {
    my $class = shift;
    my $self = {
        NAME => undef,
        AGE  => undef,
        PEERS => [],
    };
    my $closure = sub {
        my $field = shift;
        if (@_) { $self->{$field} = shift }
        return $self->{$field};
    };
    bless($closure, $class);
    return $closure;
}
sub name { &{ $_[0] }("NAME", @_[ 1 .. $#_ ] ) }
sub age { &{ $_[0] }("AGE", @_[ 1 .. $#_ ] ) }
sub peers { &{ $_[0] }("PEERS", @_[ 1 .. $#_ ] ) }
1;
```

AUTOLOAD: Proxy Methods

Autoloading is a way to intercept calls to undefined methods. An autoload routine may choose to create a new function on the fly, either loaded from disk or perhaps just eval()ed right there. This define-on-the-fly strategy is why it's called autoloading.

But that's only one possible approach. Another one is to just have the autoloaded method itself directly provide the requested service. When used in this way, you may think of autoloaded methods as "proxy" methods.

When Perl tries to call an undefined function in a particular package and that function is not defined, it looks for a function in that same package called AUTOLOAD. If one exists, it's called with the same arguments as the original function would have had. The fully-qualified name of the function is stored in that package's global variable \$AUTOLOAD. Once called, the function can do anything it would like, including defining a new function by the right name, and then doing a really fancy kind of goto right to it, erasing itself from the call stack.

AUTHOR AND COPYRIGHT

Copyright (c) 1997, 1998 Tom Christiansen All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

Acknowledgments

Thanks to Larry Wall, Roderick Schertler, Gurusamy Sarathy, Dean Roehrich, Raphael Manfredi, Brent Halsey, Greg Bacon, Brad Appleton, and many others for their helpful comments.

Perl Modules

A module is just a set of related functions in a library file, i.e., a Perl package with the same name as the file. It is specifically designed to be reusable by other modules or programs.

For example, to start a traditional, non-OO module called `Some::Module`, create a file called `Some/Module.pm` and start with this template:

Then go on to declare and use your variables in functions without any qualifications. See `Exporter` and the `perlmodlib` for details on mechanics and style issues in module creation.

Perl modules are included into your program by saying

```
use Module;
```

or

```
use Module LIST;
```

This is exactly equivalent to

```
BEGIN { require Module; import Module; }
```

or

```
BEGIN { require Module; import Module LIST; }
```

As a special case

```
use Module ();
```

is exactly equivalent to

```
BEGIN { require Module; }
```

All Perl module files have the extension `.pm`. The `use` operator assumes this so you don't have to spell out `"Module.pm"` in quotes. Module names are also capitalized unless they're functioning as pragmas; pragmas are in effect compiler directives, and are sometimes called "pragmatic modules".