NAME

perlsub - Perl subroutines

SYNOPSIS

To declare subroutines:

```
sub NAME;     # A "forward" declaration.
sub NAME(PROTO);    #  ditto, but with prototypes
sub NAME : ATTRS;    #  with attributes
sub NAME(PROTO) : ATTRS;   #  with attributes and prototypes

sub NAME BLOCK    # A declaration and a definition.
sub NAME(PROTO) BLOCK   #  ditto, but with prototypes
sub NAME : ATTRS BLOCK   #  with attributes
sub NAME(PROTO) : ATTRS BLOCK #  with prototypes and attributes
```

To define an anonymous subroutine at runtime:

```
$subref = sub BLOCK;   # no proto
$subref = sub (PROTO) BLOCK;  # with proto
$subref = sub : ATTRS BLOCK;  # with attributes
$subref = sub (PROTO) : ATTRS BLOCK; # with proto and attributes
```

To import subroutines:

```
use MODULE qw(NAME1 NAME2 NAME3);
```

To call subroutines:

```
NAME(LIST);    # & is optional with parentheses.
NAME LIST;    # Parentheses optional if predeclared/imported.
&NAME(LIST);   # Circumvent prototypes.
&NAME;    # Makes current @_ visible to called subroutine.
```

DESCRIPTION

Like many languages, Perl provides for user-defined subroutines. These may be located anywhere in the main program, loaded in from other files via the `do`, `require`, or `use` keywords, or generated on the fly using `eval` or anonymous subroutines. You can even call a function indirectly using a variable containing its name or a CODE reference.

The Perl model for function call and return values is simple: all functions are passed as parameters one single flat list of scalars, and all functions likewise return to their caller one single flat list of scalars. Any arrays or hashes in these call and return lists will collapse, losing their identities--but you may always use pass-by-reference instead to avoid this. Both call and return lists may contain as many or as few scalar elements as you'd like. (Often a function without an explicit return statement is called a subroutine, but there's really no difference from Perl's perspective.)

Any arguments passed in show up in the array `@_`. Therefore, if you called a function with two arguments, those would be stored in `$_[0]` and `$_[1]`. The array `@_` is a local array, but its elements are aliases for the actual scalar parameters. In particular, if an element `$_[0]` is updated, the corresponding argument is updated (or an error occurs if it is not updatable). If an argument is an array or hash element which did not exist when the function was called, that element is created only when (and if) it is modified or a reference to it is taken. (Some earlier versions of Perl created the element whether or not the element was assigned to.) Assigning to the whole array `@_` removes that aliasing, and does not update any arguments.

A `return` statement may be used to exit a subroutine, optionally specifying the returned value, which will be evaluated in the appropriate context (list, scalar, or void) depending on the context of the subroutine call. If you specify no return value, the subroutine returns an empty list in list context, the undefined value in scalar context, or nothing in void context. If you return one or more aggregates (arrays and hashes), these will be flattened together into one large indistinguishable list.

If no `return` is found and if the last statement is an expression, its value is returned. If the last statement is a loop control structure like a `foreach` or a `while`, the returned value is unspecified. The empty sub returns the empty list.

Perl does not have named formal parameters. In practice all you do is assign to a `my()` list of these. Variables that aren't declared to be private are global variables. For gory details on creating private variables, see *Private Variables via my()* and *Temporary Values via local()*. To create protected environments for a set of functions in a separate package (and probably a separate file), see *"Packages" in perlmod*.

Example:

```
    sub max {
my $max = shift(@_);
foreach $foo (@_) {
    $max = $foo if $max < $foo;
}
return $max;
    }
    $bestday = max($mon,$tue,$wed,$thu,$fri);
```

Example:

```
    # get a line, combining continuation lines
    #  that start with whitespace

    sub get_line {
$thisline = $lookahead;  # global variables!
LINE: while (defined($lookahead = <STDIN>)) {
    if ($lookahead =~ /^[ \t]/) {
 $thisline .= $lookahead;
    }
    else {
 last LINE;
    }
}
return $thisline;
    }

    $lookahead = <STDIN>; # get first line
    while (defined($line = get_line())) {
...
    }
```

Assigning to a list of private variables to name your arguments:

```
    sub maybeset {
my($key, $value) = @_;
$Foo{$key} = $value unless $Foo{$key};
    }
```

Because the assignment copies the values, this also has the effect of turning call-by-reference into call-by-value. Otherwise a function is free to do in-place modifications of @_ and change its caller's values.

```
    upcase_in($v1, $v2);  # this changes $v1 and $v2
    sub upcase_in {
 for (@_) { tr/a-z/A-Z/ }
    }
```

You aren't allowed to modify constants in this way, of course. If an argument were actually literal and you tried to change it, you'd take a (presumably fatal) exception. For example, this won't work:

```
    upcase_in("frederick");
```

It would be much safer if the `upcase_in()` function were written to return a copy of its parameters instead of changing them in place:

```
    ($v3, $v4) = upcase($v1, $v2);  # this doesn't change $v1 and $v2
    sub upcase {
 return unless defined wantarray;  # void context, do nothing
 my @parms = @_;
 for (@parms) { tr/a-z/A-Z/ }
   return wantarray ? @parms : $parms[0];
    }
```

Notice how this (unprototyped) function doesn't care whether it was passed real scalars or arrays. Perl sees all arguments as one big, long, flat parameter list in @_. This is one area where Perl's simple argument-passing style shines. The `upcase()` function would work perfectly well without changing the `upcase()` definition even if we fed it things like this:

```
    @newlist   = upcase(@list1, @list2);
    @newlist   = upcase( split /:/, $var );
```

Do not, however, be tempted to do this:

```
    (@a, @b)   = upcase(@list1, @list2);
```

Like the flattened incoming parameter list, the return list is also flattened on return. So all you have managed to do here is stored everything in @a and made @b empty. See *Pass by Reference* for alternatives.

A subroutine may be called using an explicit & prefix. The & is optional in modern Perl, as are parentheses if the subroutine has been predeclared. The & is *not* optional when just naming the subroutine, such as when it's used as an argument to defined() or undef(). Nor is it optional when you want to do an indirect subroutine call with a subroutine name or reference using the `&$subref()` or `&{$subref}()` constructs, although the `$subref->()` notation solves that problem. See *perlref* for more about all that.

Subroutines may be called recursively. If a subroutine is called using the & form, the argument list is optional, and if omitted, no @_ array is set up for the subroutine: the @_ array at the time of the call is visible to subroutine instead. This is an efficiency mechanism that new users may wish to avoid.

```
    &foo(1,2,3); # pass three arguments
    foo(1,2,3);  # the same

    foo();  # pass a null list
    &foo();  # the same
```

```
    &foo;  # foo() get current args, like foo(@_) !!
    foo;  # like foo() IFF sub foo predeclared, else "foo"
```

Not only does the & form make the argument list optional, it also disables any prototype checking on arguments you do provide. This is partly for historical reasons, and partly for having a convenient way to cheat if you know what you're doing. See *Prototypes* below.

Subroutines whose names are in all upper case are reserved to the Perl core, as are modules whose names are in all lower case. A subroutine in all capitals is a loosely-held convention meaning it will be called indirectly by the run-time system itself, usually due to a triggered event. Subroutines that do special, pre-defined things include `AUTOLOAD`, `CLONE`, `DESTROY` plus all functions mentioned in *perltie* and *PerlIO::via*.

The `BEGIN`, `UNITCHECK`, `CHECK`, `INIT` and `END` subroutines are not so much subroutines as named special code blocks, of which you can have more than one in a package, and which you can **not** call explicitly. See *"BEGIN, UNITCHECK, CHECK, INIT and END" in perlmod*

Private Variables via my()

Synopsis:

```
    my $foo;      # declare $foo lexically local
    my (@wid, %get);  # declare list of variables local
    my $foo = "flurp"; # declare $foo lexical, and init it
    my @oof = @bar; # declare @oof lexical, and init it
    my $x : Foo = $y; # similar, with an attribute applied
```

WARNING: The use of attribute lists on `my` declarations is still evolving. The current semantics and interface are subject to change. See *attributes* and *Attribute::Handlers*.

The `my` operator declares the listed variables to be lexically confined to the enclosing block, conditional (`if/unless/elsif/else`), loop (`for/foreach/while/until/continue`), subroutine, `eval`, or `do/require/use`'d file. If more than one value is listed, the list must be placed in parentheses. All listed elements must be legal lvalues. Only alphanumeric identifiers may be lexically scoped--magical built-ins like `$/` must currently be `local`ized with `local` instead.

Unlike dynamic variables created by the `local` operator, lexical variables declared with `my` are totally hidden from the outside world, including any called subroutines. This is true if it's the same subroutine called from itself or elsewhere--every call gets its own copy.

This doesn't mean that a `my` variable declared in a statically enclosing lexical scope would be invisible. Only dynamic scopes are cut off. For example, the `bumpx()` function below has access to the lexical $x variable because both the `my` and the `sub` occurred at the same scope, presumably file scope.

```
    my $x = 10;
    sub bumpx { $x++ }
```

An `eval()`, however, can see lexical variables of the scope it is being evaluated in, so long as the names aren't hidden by declarations within the `eval()` itself. See *perlref*.

The parameter list to my() may be assigned to if desired, which allows you to initialize your variables. (If no initializer is given for a particular variable, it is created with the undefined value.) Commonly this is used to name input parameters to a subroutine. Examples:

```
    $arg = "fred";   # "global" variable
    $n = cube_root(27);
    print "$arg thinks the root is $n\n";
 fred thinks the root is 3
```

```
    sub cube_root {
my $arg = shift;  # name doesn't matter
$arg **= 1/3;
return $arg;
    }
```

The my is simply a modifier on something you might assign to. So when you do assign to variables in its argument list, my doesn't change whether those variables are viewed as a scalar or an array. So

```
    my ($foo) = <STDIN>;  # WRONG?
    my @FOO = <STDIN>;
```

both supply a list context to the right-hand side, while

```
    my $foo = <STDIN>;
```

supplies a scalar context. But the following declares only one variable:

```
    my $foo, $bar = 1;   # WRONG
```

That has the same effect as

```
    my $foo;
    $bar = 1;
```

The declared variable is not introduced (is not visible) until after the current statement. Thus,

```
    my $x = $x;
```

can be used to initialize a new $x with the value of the old $x, and the expression

```
    my $x = 123 and $x == 123
```

is false unless the old $x happened to have the value 123.

Lexical scopes of control structures are not bounded precisely by the braces that delimit their controlled blocks; control expressions are part of that scope, too. Thus in the loop

```
    while (my $line = <>) {
        $line = lc $line;
    } continue {
        print $line;
    }
```

the scope of $line extends from its declaration throughout the rest of the loop construct (including the continue clause), but not beyond it. Similarly, in the conditional

```
    if ((my $answer = <STDIN>) =~ /^yes$/i) {
        user_agrees();
    } elsif ($answer =~ /^no$/i) {
        user_disagrees();
    } else {
chomp $answer;
        die "'$answer' is neither 'yes' nor 'no'";
    }
```

the scope of $answer extends from its declaration through the rest of that conditional, including any `elsif` and `else` clauses, but not beyond it. See *"Simple statements" in perlsyn* for information on the scope of variables in statements with modifiers.

The `foreach` loop defaults to scoping its index variable dynamically in the manner of `local`. However, if the index variable is prefixed with the keyword `my`, or if there is already a lexical by that name in scope, then a new lexical is created instead. Thus in the loop

```
for my $i (1, 2, 3) {
    some_function();
}
```

the scope of $i extends to the end of the loop, but not beyond it, rendering the value of $i inaccessible within `some_function()`.

Some users may wish to encourage the use of lexically scoped variables. As an aid to catching implicit uses to package variables, which are always global, if you say

```
use strict 'vars';
```

then any variable mentioned from there to the end of the enclosing block must either refer to a lexical variable, be predeclared via `our` or `use vars`, or else must be fully qualified with the package name. A compilation error results otherwise. An inner block may countermand this with `no strict 'vars'`.

A `my` has both a compile-time and a run-time effect. At compile time, the compiler takes notice of it. The principal usefulness of this is to quiet `use strict 'vars'`, but it is also essential for generation of closures as detailed in *perlref*. Actual initialization is delayed until run time, though, so it gets executed at the appropriate time, such as each time through a loop, for example.

Variables declared with `my` are not part of any package and are therefore never fully qualified with the package name. In particular, you're not allowed to try to make a package variable (or other global) lexical:

```
my $pack::var; # ERROR!  Illegal syntax
```

In fact, a dynamic variable (also known as package or global variables) are still accessible using the fully qualified `::` notation even while a lexical of the same name is also visible:

```
package main;
local $x = 10;
my    $x = 20;
print "$x and $::x\n";
```

That will print out `20` and `10`.

You may declare `my` variables at the outermost scope of a file to hide any such identifiers from the world outside that file. This is similar in spirit to C's static variables when they are used at the file level. To do this with a subroutine requires the use of a closure (an anonymous function that accesses enclosing lexicals). If you want to create a private subroutine that cannot be called from outside that block, it can declare a lexical variable containing an anonymous sub reference:

```
my $secret_version = '1.001-beta';
my $secret_sub = sub { print $secret_version };
&$secret_sub();
```

As long as the reference is never returned by any function within the module, no outside module can see the subroutine, because its name is not in any package's symbol table. Remember that it's not

REALLY called `$some_pack::secret_version` or anything; it's just $secret_version, unqualified and unqualifiable.

This does not work with object methods, however; all object methods have to be in the symbol table of some package to be found. See *"Function Templates" in perlref* for something of a work-around to this.

Persistent Private Variables

There are two ways to build persistent private variables in Perl 5.10. First, you can simply use the `state` feature. Or, you can use closures, if you want to stay compatible with releases older than 5.10.

Persistent variables via state()

Beginning with perl 5.9.4, you can declare variables with the `state` keyword in place of `my`. For that to work, though, you must have enabled that feature beforehand, either by using the `feature` pragma, or by using `-E` on one-liners. (see *feature*)

For example, the following code maintains a private counter, incremented each time the gimme_another() function is called:

```
use feature 'state';
sub gimme_another { state $x; return ++$x }
```

Also, since $x is lexical, it can't be reached or modified by any Perl code outside.

When combined with variable declaration, simple scalar assignment to `state` variables (as in `state $x = 42`) is executed only the first time. When such statements are evaluated subsequent times, the assignment is ignored. The behavior of this sort of assignment to non-scalar variables is undefined.

Persistent variables with closures

Just because a lexical variable is lexically (also called statically) scoped to its enclosing block, `eval`, or `do` FILE, this doesn't mean that within a function it works like a C static. It normally works more like a C auto, but with implicit garbage collection.

Unlike local variables in C or C++, Perl's lexical variables don't necessarily get recycled just because their scope has exited. If something more permanent is still aware of the lexical, it will stick around. So long as something else references a lexical, that lexical won't be freed--which is as it should be. You wouldn't want memory being free until you were done using it, or kept around once you were done. Automatic garbage collection takes care of this for you.

This means that you can pass back or save away references to lexical variables, whereas to return a pointer to a C auto is a grave error. It also gives us a way to simulate C's function statics. Here's a mechanism for giving a function private variables with both lexical scoping and a static lifetime. If you do want to create something like C's static variables, just enclose the whole function in an extra block, and put the static variable outside the function but in the block.

```
    {
my $secret_val = 0;
sub gimme_another {
    return ++$secret_val;
}
    }
    # $secret_val now becomes unreachable by the outside
    # world, but retains its value between calls to gimme_another
```

If this function is being sourced in from a separate file via `require` or `use`, then this is probably just fine. If it's all in the main program, you'll need to arrange for the `my` to be executed early, either by putting the whole block above your main program, or more likely, placing merely a `BEGIN` code block

around it to make sure it gets executed before your program starts to run:

```
    BEGIN {
 my $secret_val = 0;
 sub gimme_another {
     return ++$secret_val;
 }
    }
```

See *"BEGIN, UNITCHECK, CHECK, INIT and END" in perlmod* about the special triggered code blocks, `BEGIN`, `UNITCHECK`, `CHECK`, `INIT` and `END`.

If declared at the outermost scope (the file scope), then lexicals work somewhat like C's file statics. They are available to all functions in that same file declared below them, but are inaccessible from outside that file. This strategy is sometimes used in modules to create private variables that the whole module can see.

Temporary Values via local()

WARNING: In general, you should be using `my` instead of `local`, because it's faster and safer. Exceptions to this include the global punctuation variables, global filehandles and formats, and direct manipulation of the Perl symbol table itself. `local` is mostly used when the current value of a variable must be visible to called subroutines.

Synopsis:

```
    # localization of values

    local $foo;   # make $foo dynamically local
    local (@wid, %get);  # make list of variables local
    local $foo = "flurp"; # make $foo dynamic, and init it
    local @oof = @bar;  # make @oof dynamic, and init it

    local $hash{key} = "val"; # sets a local value for this hash entry
    local ($cond ? $v1 : $v2); # several types of lvalues support
    # localization

    # localization of symbols

    local *FH;   # localize $FH, @FH, %FH, &FH  ...
    local *merlyn = *randal; # now $merlyn is really $randal, plus
                             #     @merlyn is really @randal, etc
    local *merlyn = 'randal'; # SAME THING: promote 'randal' to *randal
    local *merlyn = \$randal;   # just alias $merlyn, not @merlyn etc
```

A `local` modifies its listed variables to be "local" to the enclosing block, `eval`, or `do FILE`--and to *any subroutine called from within that block*. A `local` just gives temporary values to global (meaning package) variables. It does *not* create a local variable. This is known as dynamic scoping. Lexical scoping is done with `my`, which works more like C's auto declarations.

Some types of lvalues can be localized as well : hash and array elements and slices, conditionals (provided that their result is always localizable), and symbolic references. As for simple variables, this creates new, dynamically scoped values.

If more than one variable or expression is given to `local`, they must be placed in parentheses. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval. This means that called subroutines can

also reference the local variable, but not the global one. The argument list may be assigned to if desired, which allows you to initialize your local variables. (If no initializer is given for a particular variable, it is created with an undefined value.)

Because `local` is a run-time operator, it gets executed each time through a loop. Consequently, it's more efficient to localize your variables outside the loop.

Grammatical note on local()

A `local` is simply a modifier on an lvalue expression. When you assign to a `local`ized variable, the `local` doesn't change whether its list is viewed as a scalar or an array. So

```
local($foo) = <STDIN>;
local @FOO = <STDIN>;
```

both supply a list context to the right-hand side, while

```
local $foo = <STDIN>;
```

supplies a scalar context.

Localization of special variables

If you localize a special variable, you'll be giving a new value to it, but its magic won't go away. That means that all side-effects related to this magic still work with the localized value.

This feature allows code like this to work :

```
# Read the whole contents of FILE in $slurp
{ local $/ = undef; $slurp = <FILE>; }
```

Note, however, that this restricts localization of some values ; for example, the following statement dies, as of perl 5.9.0, with an error *Modification of a read-only value attempted*, because the $1 variable is magical and read-only :

```
local $1 = 2;
```

Similarly, but in a way more difficult to spot, the following snippet will die in perl 5.9.0 :

```
    sub f { local $_ = "foo"; print }
    for ($1) {
 # now $_ is aliased to $1, thus is magic and readonly
 f();
    }
```

See next section for an alternative to this situation.

WARNING: Localization of tied arrays and hashes does not currently work as described. This will be fixed in a future release of Perl; in the meantime, avoid code that relies on any particular behaviour of localising tied arrays or hashes (localising individual elements is still okay). See *"Localising Tied Arrays and Hashes Is Broken" in perl58delta* for more details.

Localization of globs

The construct

```
local *name;
```

creates a whole new symbol table entry for the glob `name` in the current package. That means that all variables in its glob slot ($name, @name, %name, &name, and the `name` filehandle) are dynamically

reset.This implies, among other things, that any magic eventually carried by those variables is locally lost. In other words, saying `local */` will not have any effect on the internal value of the input record separator.

Notably, if you want to work with a brand new value of the default scalar $_, and avoid the potential problem listed above about $_ previously carrying a magic value, you should use `local *_` instead of `local $_`. As of perl 5.9.1, you can also use the lexical form of $_ (declaring it with `my $_`), which avoids completely this problem.

Localization of elements of composite types

It's also worth taking a moment to explain what happens when you `local`ize a member of a composite type (i.e. an array or hash element). In this case, the element is `local`ized *by name*. This means that when the scope of the `local()` ends, the saved value will be restored to the hash element whose key was named in the `local()`, or the array element whose index was named in the `local()`. If that element was deleted while the `local()` was in effect (e.g. by a `delete()` from a hash or a `shift()` of an array), it will spring back into existence, possibly extending an array and filling in the skipped elements with `undef`. For instance, if you say

```
%hash = ( 'This' => 'is', 'a' => 'test' );
@ary  = ( 0..5 );
{
     local($ary[5]) = 6;
     local($hash{'a'}) = 'drill';
     while (my $e = pop(@ary)) {
         print "$e . . .\n";
         last unless $e > 3;
     }
     if (@ary) {
         $hash{'only a'} = 'test';
         delete $hash{'a'};
     }
}
print join(' ', map { "$_ $hash{$_}" } sort keys %hash),".\n";
print "The array has ",scalar(@ary)," elements: ",
      join(', ', map { defined $_ ? $_ : 'undef' } @ary),"\n";
```

Perl will print

```
6 . . .
4 . . .
3 . . .
This is a test only a test.
The array has 6 elements: 0, 1, 2, undef, undef, 5
```

The behavior of local() on non-existent members of composite types is subject to change in future.

Lvalue subroutines

WARNING: Lvalue subroutines are still experimental and the implementation may change in future versions of Perl.

It is possible to return a modifiable value from a subroutine. To do this, you have to declare the subroutine to return an lvalue.

```
    my $val;
    sub canmod : lvalue {
 # return $val; this doesn't work, don't say "return"
 $val;
```

```
    }
    sub nomod {
 $val;
    }

    canmod() = 5;   # assigns to $val
    nomod()  = 5;   # ERROR
```

The scalar/list context for the subroutine and for the right-hand side of assignment is determined as if the subroutine call is replaced by a scalar. For example, consider:

```
    data(2,3) = get_data(3,4);
```

Both subroutines here are called in a scalar context, while in:

```
    (data(2,3)) = get_data(3,4);
```

and in:

```
    (data(2),data(3)) = get_data(3,4);
```

all the subroutines are called in a list context.

Lvalue subroutines are EXPERIMENTAL

They appear to be convenient, but there are several reasons to be circumspect.

You can't use the return keyword, you must pass out the value before falling out of subroutine scope. (see comment in example above). This is usually not a problem, but it disallows an explicit return out of a deeply nested loop, which is sometimes a nice way out.

They violate encapsulation. A normal mutator can check the supplied argument before setting the attribute it is protecting, an lvalue subroutine never gets that chance. Consider;

```
    my $some_array_ref = []; # protected by mutators ??

    sub set_arr {   # normal mutator
 my $val = shift;
 die("expected array, you supplied ", ref $val)
    unless ref $val eq 'ARRAY';
 $some_array_ref = $val;
    }
    sub set_arr_lv : lvalue { # lvalue mutator
 $some_array_ref;
    }

    # set_arr_lv cannot stop this !
    set_arr_lv() = { a => 1 };
```

Passing Symbol Table Entries (typeglobs)

WARNING: The mechanism described in this section was originally the only way to simulate pass-by-reference in older versions of Perl. While it still works fine in modern versions, the new reference mechanism is generally easier to work with. See below.

Sometimes you don't want to pass the value of an array to a subroutine but rather the name of it, so that the subroutine can modify the global copy of it rather than working with a local copy. In perl you can refer to all objects of a particular name by prefixing the name with a star: `*foo`. This is often known as a "typeglob", because the star on the front can be thought of as a wildcard match for all the

funny prefix characters on variables and subroutines and such.

When evaluated, the typeglob produces a scalar value that represents all the objects of that name, including any filehandle, format, or subroutine. When assigned to, it causes the name mentioned to refer to whatever * value was assigned to it. Example:

```
    sub doubleary {
local(*someary) = @_;
foreach $elem (@someary) {
    $elem *= 2;
}
    }
    doubleary(*foo);
    doubleary(*bar);
```

Scalars are already passed by reference, so you can modify scalar arguments without using this mechanism by referring explicitly to `$_[0]` etc. You can modify all the elements of an array by passing all the elements as scalars, but you have to use the * mechanism (or the equivalent reference mechanism) to `push`, `pop`, or change the size of an array. It will certainly be faster to pass the typeglob (or reference).

Even if you don't want to modify an array, this mechanism is useful for passing multiple arrays in a single LIST, because normally the LIST mechanism will merge all the array values so that you can't extract out the individual arrays. For more on typeglobs, see *"Typeglobs and Filehandles" in perldata*.

When to Still Use local()

Despite the existence of `my`, there are still three places where the `local` operator still shines. In fact, in these three places, you *must* use `local` instead of `my`.

1. You need to give a global variable a temporary value, especially `$_`.

   The global variables, like `@ARGV` or the punctuation variables, must be `local`ized with `local()`. This block reads in */etc/motd*, and splits it up into chunks separated by lines of equal signs, which are placed in `@Fields`.

   ```
       {
    local @ARGV = ("/etc/motd");
          local $/ = undef;
          local $_ = <>;
    @Fields = split /^\s*=+\s*$/;
       }
   ```

   It particular, it's important to `local`ize `$_` in any routine that assigns to it. Look out for implicit assignments in `while` conditionals.

2. You need to create a local file or directory handle or a local function.

   A function that needs a filehandle of its own must use `local()` on a complete typeglob. This can be used to create new symbol table entries:

   ```
       sub ioqueue {
           local  (*READER, *WRITER);    # not my!
           pipe    (READER,  WRITER)     or die "pipe: $!";
           return (*READER, *WRITER);
       }
       ($head, $tail) = ioqueue();
   ```

   See the Symbol module for a way to create anonymous symbol table entries.

   Because assignment of a reference to a typeglob creates an alias, this can be used to create what is effectively a local function, or at least, a local alias.

```
    {
        local *grow = \&shrink; # only until this block exists
        grow();                 # really calls shrink()
 move();   # if move() grow()s, it shrink()s too
    }
    grow();   # get the real grow() again
```

See *"Function Templates" in perlref* for more about manipulating functions by name in this way.

3. You want to temporarily change just one element of an array or hash.

   You can `local`ize just one element of an aggregate. Usually this is done on dynamics:

```
    {
 local $SIG{INT} = 'IGNORE';
 funct();       # uninterruptible
    }
    # interruptibility automatically restored here
```

   But it also works on lexically declared aggregates. Prior to 5.005, this operation could on occasion misbehave.

Pass by Reference

If you want to pass more than one array or hash into a function--or return them from it--and have them maintain their integrity, then you're going to have to use an explicit pass-by-reference. Before you do that, you need to understand references as detailed in *perlref*. This section may not make much sense to you otherwise.

Here are a few simple examples. First, let's pass in several arrays to a function and have it `pop` all of then, returning a new list of all their former last elements:

```
    @tailings = popmany ( \@a, \@b, \@c, \@d );

    sub popmany {
 my $aref;
 my @retlist = ();
 foreach $aref ( @_ ) {
     push @retlist, pop @$aref;
 }
 return @retlist;
    }
```

Here's how you might write a function that returns a list of keys occurring in all the hashes passed to it:

```
    @common = inter( \%foo, \%bar, \%joe );
    sub inter {
 my ($k, $href, %seen); # locals
 foreach $href (@_) {
     while ( $k = each %$href ) {
  $seen{$k}++;
     }
 }
 return grep { $seen{$_} == @_ } keys %seen;
    }
```

So far, we're using just the normal list return mechanism. What happens if you want to pass or return

a hash? Well, if you're using only one of them, or you don't mind them concatenating, then the normal calling convention is ok, although a little expensive.

Where people get into trouble is here:

```
    (@a, @b) = func(@c, @d);
or
    (%a, %b) = func(%c, %d);
```

That syntax simply won't work. It sets just @a or %a and clears the @b or %b. Plus the function didn't get passed into two separate arrays or hashes: it got one long list in @_, as always.

If you can arrange for everyone to deal with this through references, it's cleaner code, although not so nice to look at. Here's a function that takes two array references as arguments, returning the two array elements in order of how many elements they have in them:

```
    ($aref, $bref) = func(\@c, \@d);
    print "@$aref has more than @$bref\n";
    sub func {
 my ($cref, $dref) = @_;
 if (@$cref > @$dref) {
     return ($cref, $dref);
 } else {
     return ($dref, $cref);
 }
    }
```

It turns out that you can actually do this also:

```
    (*a, *b) = func(\@c, \@d);
    print "@a has more than @b\n";
    sub func {
 local (*c, *d) = @_;
 if (@c > @d) {
     return (\@c, \@d);
 } else {
     return (\@d, \@c);
 }
    }
```

Here we're using the typeglobs to do symbol table aliasing. It's a tad subtle, though, and also won't work if you're using my variables, because only globals (even in disguise as `locals`) are in the symbol table.

If you're passing around filehandles, you could usually just use the bare typeglob, like `*STDOUT`, but typeglobs references work, too. For example:

```
    splutter(\*STDOUT);
    sub splutter {
 my $fh = shift;
 print $fh "her um well a hmmm\n";
    }

    $rec = get_rec(\*STDIN);
    sub get_rec {
 my $fh = shift;
 return scalar <$fh>;
```

```
    }
```

If you're planning on generating new filehandles, you could do this. Notice to pass back just the bare *FH, not its reference.

```
    sub openit {
 my $path = shift;
 local *FH;
 return open (FH, $path) ? *FH : undef;
    }
```

Prototypes

Perl supports a very limited kind of compile-time argument checking using function prototyping. If you declare

```
    sub mypush (\@@)
```

then `mypush()` takes arguments exactly like `push()` does. The function declaration must be visible at compile time. The prototype affects only interpretation of new-style calls to the function, where new-style is defined as not using the `&` character. In other words, if you call it like a built-in function, then it behaves like a built-in function. If you call it like an old-fashioned subroutine, then it behaves like an old-fashioned subroutine. It naturally falls out from this rule that prototypes have no influence on subroutine references like `\&foo` or on indirect subroutine calls like `&{$subref}` or `$subref->()`.

Method calls are not influenced by prototypes either, because the function to be called is indeterminate at compile time, since the exact code called depends on inheritance.

Because the intent of this feature is primarily to let you define subroutines that work like built-in functions, here are prototypes for some other functions that parse almost exactly like the corresponding built-in.

```
    Declared as   Called as

    sub mylink ($$)      mylink $old, $new
    sub myvec ($$$)      myvec $var, $offset, 1
    sub myindex ($$;$)      myindex &getstring, "substr"
    sub mysyswrite ($$$;$)   mysyswrite $buf, 0, length($buf) - $off, $off
    sub myreverse (@)      myreverse $a, $b, $c
    sub myjoin ($@)      myjoin ":", $a, $b, $c
    sub mypop (\@)      mypop @array
    sub mysplice (\@$$@)     mysplice @array, @array, 0, @pushme
    sub mykeys (\%)      mykeys %{$hashref}
    sub myopen (*;$)      myopen HANDLE, $name
    sub mypipe (**)      mypipe READHANDLE, WRITEHANDLE
    sub mygrep (&@)      mygrep { /foo/ } $a, $b, $c
    sub myrand (;$)      myrand 42
    sub mytime ()      mytime
```

Any backslashed prototype character represents an actual argument that absolutely must start with that character. The value passed as part of `@_` will be a reference to the actual argument given in the subroutine call, obtained by applying `\` to that argument.

You can also backslash several argument types simultaneously by using the `\[]` notation:

```
    sub myref (\[$@%&*])
```

will allow calling myref() as

```
    myref $var
    myref @array
    myref %hash
    myref &sub
    myref *glob
```

and the first argument of myref() will be a reference to a scalar, an array, a hash, a code, or a glob.

Unbackslashed prototype characters have special meanings. Any unbackslashed @ or % eats all remaining arguments, and forces list context. An argument represented by $ forces scalar context. An & requires an anonymous subroutine, which, if passed as the first argument, does not require the sub keyword or a subsequent comma.

A * allows the subroutine to accept a bareword, constant, scalar expression, typeglob, or a reference to a typeglob in that slot. The value will be available to the subroutine either as a simple scalar, or (in the latter two cases) as a reference to the typeglob. If you wish to always convert such arguments to a typeglob reference, use Symbol::qualify_to_ref() as follows:

```
    use Symbol 'qualify_to_ref';

    sub foo (*) {
 my $fh = qualify_to_ref(shift, caller);
 ...
    }
```

A semicolon (;) separates mandatory arguments from optional arguments. It is redundant before @ or %, which gobble up everything else.

As the last character of a prototype, or just before a semicolon, you can use _ in place of $: if this argument is not provided, $_ will be used instead.

Note how the last three examples in the table above are treated specially by the parser. mygrep() is parsed as a true list operator, myrand() is parsed as a true unary operator with unary precedence the same as rand(), and mytime() is truly without arguments, just like time(). That is, if you say

```
    mytime +2;
```

you'll get mytime() + 2, not mytime(2), which is how it would be parsed without a prototype.

The interesting thing about & is that you can generate new syntax with it, provided it's in the initial position:

```
    sub try (&@) {
 my($try,$catch) = @_;
 eval { &$try };
 if ($@) {
     local $_ = $@;
     &$catch;
 }
    }
    sub catch (&) { $_[0] }

    try {
 die "phooey";
    } catch {
```

```
/phooey/ and print "unphooey\n";
   };
```

That prints "unphooey". (Yes, there are still unresolved issues having to do with visibility of @_. I'm ignoring that question for the moment. (But note that if we make @_ lexically scoped, those anonymous subroutines can act like closures... (Gee, is this sounding a little Lispish? (Never mind.))))

And here's a reimplementation of the Perl grep operator:

```
   sub mygrep (&@) {
my $code = shift;
my @result;
foreach $_ (@_) {
    push(@result, $_) if &$code;
}
@result;
   }
```

Some folks would prefer full alphanumeric prototypes. Alphanumerics have been intentionally left out of prototypes for the express purpose of someday in the future adding named, formal parameters. The current mechanism's main goal is to let module writers provide better diagnostics for module users. Larry feels the notation quite understandable to Perl programmers, and that it will not intrude greatly upon the meat of the module, nor make it harder to read. The line noise is visually encapsulated into a small pill that's easy to swallow.

If you try to use an alphanumeric sequence in a prototype you will generate an optional warning - "Illegal character in prototype...". Unfortunately earlier versions of Perl allowed the prototype to be used as long as its prefix was a valid prototype. The warning may be upgraded to a fatal error in a future version of Perl once the majority of offending code is fixed.

It's probably best to prototype new functions, not retrofit prototyping into older ones. That's because you must be especially careful about silent impositions of differing list versus scalar contexts. For example, if you decide that a function should take just one parameter, like this:

```
   sub func ($) {
my $n = shift;
print "you gave me $n\n";
   }
```

and someone has been calling it with an array or expression returning a list:

```
   func(@foo);
   func( split /:/ );
```

Then you've just supplied an automatic scalar in front of their argument, which can be more than a bit surprising. The old @foo which used to hold one thing doesn't get passed in. Instead, func() now gets passed in a 1; that is, the number of elements in @foo. And the split gets called in scalar context so it starts scribbling on your @_ parameter list. Ouch!

This is all very powerful, of course, and should be used only in moderation to make the world a better place.

Constant Functions

Functions with a prototype of () are potential candidates for inlining. If the result after optimization and constant folding is either a constant or a lexically-scoped scalar which has no other references, then it will be used in place of function calls made without &. Calls made using & are never inlined. (See *constant.pm* for an easy way to declare most constants.)

The following functions would all be inlined:

```
    sub pi ()  { 3.14159 }  # Not exact, but close.
    sub PI ()  { 4 * atan2 1, 1 } # As good as it gets,
      # and it's inlined, too!
    sub ST_DEV () { 0 }
    sub ST_INO () { 1 }

    sub FLAG_FOO () { 1 << 8 }
    sub FLAG_BAR () { 1 << 9 }
    sub FLAG_MASK () { FLAG_FOO | FLAG_BAR }

    sub OPT_BAZ () { not (0x1B58 & FLAG_MASK) }

    sub N () { int(OPT_BAZ) / 3 }

    sub FOO_SET () { 1 if FLAG_MASK & FLAG_FOO }
```

Be aware that these will not be inlined; as they contain inner scopes, the constant folding doesn't reduce them to a single constant:

```
    sub foo_set () { if (FLAG_MASK & FLAG_FOO) { 1 } }

    sub baz_val () {
 if (OPT_BAZ) {
     return 23;
 }
 else {
     return 42;
 }
    }
```

If you redefine a subroutine that was eligible for inlining, you'll get a mandatory warning. (You can use this warning to tell whether or not a particular subroutine is considered constant.) The warning is considered severe enough not to be optional because previously compiled invocations of the function will still be using the old value of the function. If you need to be able to redefine the subroutine, you need to ensure that it isn't inlined, either by dropping the `()` prototype (which changes calling semantics, so beware) or by thwarting the inlining mechanism in some other way, such as

```
    sub not_inlined () {
     23 if $];
    }
```

Overriding Built-in Functions

Many built-in functions may be overridden, though this should be tried only occasionally and for good reason. Typically this might be done by a package attempting to emulate missing built-in functionality on a non-Unix system.

Overriding may be done only by importing the name from a module at compile time--ordinary predeclaration isn't good enough. However, the `use subs` pragma lets you, in effect, predeclare subs via the import syntax, and these names may then override built-in ones:

```
    use subs 'chdir', 'chroot', 'chmod', 'chown';
    chdir $somewhere;
    sub chdir { ... }
```

To unambiguously refer to the built-in form, precede the built-in name with the special package qualifier `CORE::`. For example, saying `CORE::open()` always refers to the built-in `open()`, even if the current package has imported some other subroutine called `&open()` from elsewhere. Even though it looks like a regular function call, it isn't: you can't take a reference to it, such as the incorrect `\&CORE::open` might appear to produce.

Library modules should not in general export built-in names like `open` or `chdir` as part of their default `@EXPORT` list, because these may sneak into someone else's namespace and change the semantics unexpectedly. Instead, if the module adds that name to `@EXPORT_OK`, then it's possible for a user to import the name explicitly, but not implicitly. That is, they could say

```
use Module 'open';
```

and it would import the `open` override. But if they said

```
use Module;
```

they would get the default imports without overrides.

The foregoing mechanism for overriding built-in is restricted, quite deliberately, to the package that requests the import. There is a second method that is sometimes applicable when you wish to override a built-in everywhere, without regard to namespace boundaries. This is achieved by importing a sub into the special namespace `CORE::GLOBAL::`. Here is an example that quite brazenly replaces the `glob` operator with something that understands regular expressions.

```
    package REGlob;
    require Exporter;
    @ISA = 'Exporter';
    @EXPORT_OK = 'glob';

    sub import {
my $pkg = shift;
return unless @_;
my $sym = shift;
my $where = ($sym =~ s/^GLOBAL_// ? 'CORE::GLOBAL' : caller(0));
$pkg->export($where, $sym, @_);
    }

    sub glob {
my $pat = shift;
my @got;
if (opendir my $d, '.') {
    @got = grep /$pat/, readdir $d;
    closedir $d;
}
return @got;
    }
    1;
```

And here's how it could be (ab)used:

```
    #use REGlob 'GLOBAL_glob';     # override glob() in ALL namespaces
    package Foo;
    use REGlob 'glob';      # override glob() in Foo:: only
    print for <^[a-z_]+\.pm\$>;     # show all pragmatic modules
```

The initial comment shows a contrived, even dangerous example. By overriding `glob` globally, you would be forcing the new (and subversive) behavior for the `glob` operator for *every* namespace, without the complete cognizance or cooperation of the modules that own those namespaces. Naturally, this should be done with extreme caution--if it must be done at all.

The `REGlob` example above does not implement all the support needed to cleanly override perl's `glob` operator. The built-in `glob` has different behaviors depending on whether it appears in a scalar or list context, but our `REGlob` doesn't. Indeed, many perl built-in have such context sensitive behaviors, and these must be adequately supported by a properly written override. For a fully functional example of overriding `glob`, study the implementation of `File::DosGlob` in the standard library.

When you override a built-in, your replacement should be consistent (if possible) with the built-in native syntax. You can achieve this by using a suitable prototype. To get the prototype of an overridable built-in, use the `prototype` function with an argument of `"CORE::builtin_name"` (see *"prototype" in perlfunc*).

Note however that some built-ins can't have their syntax expressed by a prototype (such as `system` or `chomp`). If you override them you won't be able to fully mimic their original syntax.

The built-ins `do`, `require` and `glob` can also be overridden, but due to special magic, their original syntax is preserved, and you don't have to define a prototype for their replacements. (You can't override the `do BLOCK` syntax, though).

`require` has special additional dark magic: if you invoke your `require` replacement as `require Foo::Bar`, it will actually receive the argument `"Foo/Bar.pm"` in `@_`. See *"require" in perlfunc*.

And, as you'll have noticed from the previous example, if you override `glob`, the `<*>` glob operator is overridden as well.

In a similar fashion, overriding the `readline` function also overrides the equivalent I/O operator `<FILEHANDLE>`. Also, overriding `readpipe` also overrides the operators `` `` and `qx//`.

Finally, some built-ins (e.g. `exists` or `grep`) can't be overridden.

Autoloading

If you call a subroutine that is undefined, you would ordinarily get an immediate, fatal error complaining that the subroutine doesn't exist. (Likewise for subroutines being used as methods, when the method doesn't exist in any base class of the class's package.) However, if an `AUTOLOAD` subroutine is defined in the package or packages used to locate the original subroutine, then that `AUTOLOAD` subroutine is called with the arguments that would have been passed to the original subroutine. The fully qualified name of the original subroutine magically appears in the global $AUTOLOAD variable of the same package as the `AUTOLOAD` routine. The name is not passed as an ordinary argument because, er, well, just because, that's why. (As an exception, a method call to a nonexistent `import` or `unimport` method is just skipped instead.)

Many `AUTOLOAD` routines load in a definition for the requested subroutine using eval(), then execute that subroutine using a special form of goto() that erases the stack frame of the `AUTOLOAD` routine without a trace. (See the source to the standard module documented in *AutoLoader*, for example.) But an `AUTOLOAD` routine can also just emulate the routine and never define it. For example, let's pretend that a function that wasn't defined should just invoke `system` with those arguments. All you'd do is:

```
    sub AUTOLOAD {
my $program = $AUTOLOAD;
$program =~ s/.*::/;
system($program, @_);
    }
    date();
    who('am', 'i');
```

```
ls('-l');
```

In fact, if you predeclare functions you want to call that way, you don't even need parentheses:

```
use subs qw(date who ls);
date;
who "am", "i";
ls '-l';
```

A more complete example of this is the standard Shell module, which can treat undefined subroutine calls as calls to external programs.

Mechanisms are available to help modules writers split their modules into autoloadable files. See the standard AutoLoader module described in *AutoLoader* and in *AutoSplit*, the standard SelfLoader modules in *SelfLoader*, and the document on adding C functions to Perl code in *perlxs*.

Subroutine Attributes

A subroutine declaration or definition may have a list of attributes associated with it. If such an attribute list is present, it is broken up at space or colon boundaries and treated as though a `use attributes` had been seen. See *attributes* for details about what attributes are currently supported. Unlike the limitation with the obsolescent `use attrs`, the `sub : ATTRLIST` syntax works to associate the attributes with a pre-declaration, and not just with a subroutine definition.

The attributes must be valid as simple identifier names (without any punctuation other than the '_' character). They may have a parameter list appended, which is only checked for whether its parentheses ('(',')') nest properly.

Examples of valid syntax (even though the attributes are unknown):

```
sub fnord (&\%) : switch(10,foo(7,3))  :  expensive;
sub plugh () : Ugly('\(") :Bad;
sub xyzzy : _5x5 { ... }
```

Examples of invalid syntax:

```
sub fnord : switch(10,foo(); # ()-string not balanced
sub snoid : Ugly('(');   # ()-string not balanced
sub xyzzy : 5x5;    # "5x5" not a valid identifier
sub plugh : Y2::north;   # "Y2::north" not a simple identifier
sub snurt : foo + bar;   # "+" not a colon or space
```

The attribute list is passed as a list of constant strings to the code which associates them with the subroutine. In particular, the second example of valid syntax above currently looks like this in terms of how it's parsed and invoked:

```
use attributes __PACKAGE__, \&plugh, q[Ugly('\(")], 'Bad';
```

For further details on attribute lists and their manipulation, see *attributes* and *Attribute::Handlers*.

SEE ALSO

See *"Function Templates" in perlref* for more about references and closures. See *perlxs* if you'd like to learn about calling C subroutines from Perl. See *perlembed* if you'd like to learn about calling Perl subroutines from C. See *perlmod* to learn about bundling up your functions in separate files. See *perlmodlib* to learn what library modules come standard on your system. See *perltoot* to learn how to make object method calls.