A *function* (or *subroutine*) in Perl is a discrete, encapsulated unit of behavior. A program is a collection of little black boxes where the interaction of these functions governs the control flow of the program. A function may have a name. It may consume incoming information. It may produce outgoing information.

Functions are a prime mechanism for organizing code into similar groups, identifying individual pieces by name, and providing reusable units of behavior.

Declaring Functions

Use the `sub` builtin to declare a function:

```
B<sub> greet_me  { ... }
```

Now you can invoke `greet_me()` from anywhere within your program.

Just as you may *declare* a lexical variable but leave its value undefined, you may declare a function without defining it. A *forward declaration* tells Perl to record that a named function exists. You may define it later:

```
sub greet_sun;
```

Invoking Functions

Use postfix (*fixity*) parentheses to invoke a named function. Any arguments to the function may go within the parentheses:

```
greet_me( 'Jack', 'Tuxie', 'Brad' );
greet_me( 'Snowy' );
greet_me();
```

While these parentheses are not strictly necessary for these examples--even with `strict` enabled--they provide clarity to human readers as well as Perl's parser. When in doubt, use them.

Function arguments can be arbitrary expressions--including variables and function calls:

```
greet_me( $name );
greet_me( @authors );
greet_me( %editors );
greet_me( get_readers() );
```

... though Perl's default parameter handling sometimes surprises novices.

Function Parameters

A function receives its parameters in a single array, `@_` (*default_array_variables*). When you invoke a function, Perl *flattens* all provided arguments into a single list. The function must either unpack its parameters into variables or operate on `@_` directly:

```
sub greet_one {
    B<my ($name) = @_>;
    say "Hello, $name!";
    }

    sub greet_all {
        say "Hello, B<$_>!" for B<@_>;
    }
```

`@_` behaves as a normal array. Most Perl functions `shift` off parameters or use list assignment, but you may also access individual elements by index:

```
    sub greet_one_shift {
        B<my $name = shift>;
        say "Hello, $name!";
    }

    sub greet_two_list_assignment {
        my ($hero, $sidekick) = @_;
        say "Well if it isn't $hero and $sidekick. Welcome!";
    }

    sub greet_one_indexed {
        B<my $name = $_[0]>;
        say "Hello, $name!";

        # or, less clear
        say "Hello, $_[0]!";
    }
```

You may also `unshift`, `push`, `pop`, `splice`, and use slices of `@_`. Remember that the array builtins use `@_` as the default operand *within functions*, so that `my $name = shift;` works. Take advantage of this idiom.

To access a single scalar parameter from `@_`, use `shift`, an index of `@_`, or lvalue list context parentheses. Otherwise, Perl will happily evaluate `@_` in scalar context for you and assign the number of parameters passed:

```
    sub bad_greet_one {
        B<my $name = @_>;  # buggy
        say "Hello, $name; you look numeric today!"
    }
```

List assignment of multiple parameters is often clearer than multiple lines of `shift`. Compare:

```
    my $left_value  = shift;
    my $operation   = shift;
    my $right_value = shift;
```

... to:

```
    my ($left_value, $operation, $right_value) = @_;
```

The latter is simpler to read. As a side benefit, it has better runtime performance, though you're unlikely to notice.

Occasionally you may see code which extracts parameters from `@_` and passes the rest to another function:

```
    sub delegated_method {
        my $self = B<shift>;
        say 'Calling delegated_method()'

        $self->delegate->delegated_method( B<@_> );
    }
```

Use `shift` when your function needs only a single parameter. Use list assignment when accessing multiple parameters.

Real Function Signatures

Perl 5.20 added built-in function signatures as an experimental feature. "Experimental" means that they may change or even go away in future releases of Perl, so you need to enable them to signal that you accept the possibility of rewriting code.

```
    use experimental 'signatures';
```

With that disclaimer in place, you can now write:

```
    sub greet_one(B<$name>) {
        say "Hello, $name!";
    }
```

... which is equivalent to writing:

```
    sub greet_one {
        die "Too many arguments for subroutine" if @_ < 1;
        die "Too few arguments for subroutine" if @_ > 1;
        my $name = shift;
        say "Hello, $name!";
    }
```

You can make `$name` an optional variable by assigning it a default value:

```
    sub greet_one(B<$name = 'Bruce'>) {
        say "Hello, $name!";
    }
```

... in which case writing `greet_one( 'Bruce' )` and `greet_one()` will both ignore Batman's crime-fighting identity.

You may use aggregate arguments at the end of a signature:

```
    sub greet_all(B<$leader, @everyone>) {
        say "Hello, $leader!";
        say "Hi also, $_." for @everyone;
    }

    sub make_nested_hash(B<$name, %pairs>) {
        return { $name => \%pairs };
    }
```

... or indicate that a function expects *no* arguments:

```
    sub no_gifts_please() {
        say 'I have too much stuff already.'
    }
```

... which means that you'll get the `Too many arguments for subroutine` exception by calling that function with arguments.

These experimental signatures have more features than discussed here. As you get beyond basic positional parameters, the possibility of incompatible changes in future versions of Perl increases, however. See `perldoc perlsub`'s "Signatures" section for more details, especially in newer versions of Perl.

Signatures aren't your only options. Several CPAN distributions extend Perl's parameter handling with additional syntax and options. `Method::Signatures` works as far back as Perl 5.8. `Kavorka` works with Perl 5.14 and newer.

Despite the experimental nature of function signatures--or the additional dependencies of the CPAN modules--all of these options can make your code a little shorter and a little clearer both to read and to write. By all means experiment with these options to find out what works best for you and your team. Even sticking with simple positional parameters can improve your work.

Flattening

List flattening into `@_` happens on the caller side of a function call. Passing a hash as an argument produces a list of key/value pairs:

```
my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
    Rosie   => 'dog',
);

show_pets( %pet_names_and_types );

sub show_pets {
    my %pets = @_;

    while (my ($name, $type) = each %pets) {
        say "$name is a $type";
    }
}
```

When Perl flattens `%pet_names_and_types` into a list, the order of the key/value pairs from the hash will vary, but the list will always contain a key immediately followed by its value. Hash assignment inside `show_pets()` works the same way as the explicit assignment to `%pet_names_and_types`.

This flattening is often useful, but beware of mixing scalars with flattened aggregates in parameter lists. To write a `show_pets_of_type()` function, where one parameter is the type of pet to display, pass that type as the *first* parameter (or use `pop` to remove it from the end of `@_`, if you like to confuse people):

```
sub show_pets_by_type {
    B<my ($type, %pets) = @_>;

    while (my ($name, $species) = each %pets) {
        B<next unless $species eq $type;>
        say "$name is a $species";
    }
}
```

```
my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
    Rosie   => 'dog',
);

show_pets_by_type( 'dog',   %pet_names_and_types );
show_pets_by_type( 'cat',   %pet_names_and_types );
show_pets_by_type( 'moose', %pet_names_and_types );
```

With experimental function signatures, you could write:

```
sub show_pets_by_type($type, %pets) {
    ...
}
```

Slurping

List assignment with an aggregate is always greedy, so assigning to `%pets` *slurps* all of the remaining values from `@_`. If the `$type` parameter came at the end of `@_`, Perl would warn about assigning an odd number of elements to the hash. You *could* work around that:

```
sub show_pets_by_type {
    B<my $type = pop;>
    B<my %pets = @_;>

    ...
}
```

... at the expense of clarity. The same principle applies when assigning to an array as a parameter. Use references (*references*) to avoid unwanted aggregate flattening.

Aliasing

`@_` contains a subtlety; it *aliases* function arguments. In other words, if you access `@_` directly, you can modify the arguments passed to the function:

```
sub modify_name {
    $_[0] = reverse $_[0];
}

my $name = 'Orange';
modify_name( $name );
say $name;

# prints C<egnarO>
```

Modify an element of `@_` directly and you will modify the original argument. Be cautious and unpack `@_` rigorously--or document the modification carefully.

Functions and Namespaces

Every function has a containing namespace (*packages*). Functions in an undeclared namespace--functions not declared within the scope of an explicit `package` statement--exist in the `main` namespace. You may also declare a function within another namespace by prefixing its name:

```
sub B<Extensions::Math::>add { ... }
```

This will create the namespace as necessary and then declare the function within it. Remember that Perl packages are open for modification at any point--even while your program is running. Perl will issue a warning if you declare multiple functions with the same name in a single namespace.

Refer to other functions within the same namespace with their short names. Use a fully-qualified name to invoke a function in another namespace:

```
package main;

Extensions::Math::add( $scalar, $vector );
```

Remember, functions are *visible* outside of their own namespaces through their fully-qualified names. Alternately, you may import names from other namespaces.

Perl 5.18 added an experimental feature to declare functions lexically. They're visible only within lexical scopes after declaration. See the "Lexical Subroutines" section of `perldoc perlsub` for more details.

Importing

When loading a module with the `use` builtin (*modules*), Perl automatically calls a method named `import()`. Modules can provide their own `import()` method which makes some or all defined symbols available to the calling package. Any arguments after the name of the module in the `use` statement get passed to the module's `import()` method. Thus:

```
use strict;
```

... loads the *strict.pm* module and calls `strict->import()` with no arguments, while:

```
use strict 'refs';
use strict qw( subs vars );
```

... loads the *strict.pm* module, calls `strict->import( 'refs' )`, then calls `strict->import( 'subs', vars' )`.

`use` has special behavior with regard to `import()`, but you may call `import()` directly. The `use` example is equivalent to:

```
BEGIN {
    require strict;
    strict->import( 'refs' );
    strict->import( qw( subs vars ) );
}
```

The `use` builtin adds an implicit `BEGIN` block around these statements so that the `import()` call happens *immediately* after the parser has compiled the entire `use` statement. This ensures that the parser knows about any symbols imported by `strict` before it compiles the rest of the program. Otherwise, any functions *imported* from other modules but not *declared* in the current file would look like barewords, and would violate `strict`, for example.

Of course, `strict` is a pragma (*pragmas*), so it has other effects.

Reporting Errors

Use the `caller` builtin to inspect a function's calling context. When passed no arguments, `caller` returns a list containing the name of the calling package, the name of the file containing the call, and the line number of the file on which the call occurred:

```
package main;

main();

sub main {
    show_call_information();
}

sub show_call_information {
    my ($package, $file, $line) = caller();
    say "Called from $package in $file:$line";
}
```

The full call chain is available for inspection. Pass a single integer argument *n* to `caller()` to inspect the caller of the caller of the caller *n* times back. Within `show_call_information()`, `caller(0)` returns information about the call from `main()`. `caller(1)` returns information about the call from the start of the program.

This optional argument also tells `caller` to provide additional return values, including the name of the function and the context of the call:

```
sub show_call_information {
    my ($package, $file, $lineB<, $func>) = caller(B<0>);
    say "Called B<$func> from $package in $file:$line";
}
```

The standard `Carp` module uses `caller` to enhance error and warning messages. When used in place of `die` in library code, `croak()` throws an exception from the point of view of its caller. `carp()` reports a warning from the file and line number of its caller (*producing_warnings*).

Use `caller` (or `Carp`) when validating parameters or preconditions of a function to indicate that whatever called the function did so erroneously.

Validating Arguments

While Perl does its best to do what you mean, it offers few native ways to test the validity of arguments provided to a function. Evaluate `@_` in scalar context to check that the *number* of parameters passed to a function is correct:

```
sub add_numbers {
    croak 'Expected two numbers, received: ' . @_
        unless @_ == 2;

    ...
}
```

This validation reports any parameter count error from the point of view of its caller, thanks to the use of `croak`.

Type checking is more difficult, because of Perl's operator-oriented type conversions (*context_philosophy*). If you want additional safety of function parameters, see CPAN modules such as `Params::Validate`.

Advanced Functions

Functions are the foundation of many advanced Perl features.

Context Awareness

Perl's builtins know whether you've invoked them in void, scalar, or list context. So too can your functions. The `wantarray` builtin returns `undef` to signify void context, a false value to signify scalar context, and a true value to signify list context. Yes, it's misnamed; see `perldoc -f wantarray` for proof.

```
sub context_sensitive {
    my $context = wantarray();

    return qw( List context )   if         $context;
    say    'Void context'   unless defined $context;
    return 'Scalar context' unless         $context;
}

context_sensitive();
say my $scalar = context_sensitive();
say context_sensitive();
```

This can be useful for functions which might produce expensive return values to avoid doing so in void context. Some idiomatic functions return a list in list context and the first element of the list or an array reference in scalar context. However, there exists no single best recommendation for the use of `wantarray`. Sometimes it's clearer to write separate and unambiguous functions, such as `get_all_toppings()` and `get_next_topping()`.

Robin Houston's `Want` and Damian Conway's `Contextual::Return` distributions from the CPAN offer many possibilities for writing powerful context-aware interfaces.

Recursion

Suppose you want to find an element in a sorted array. You *could* iterate through every element of the array individually, looking for the target, but on average, you'll have to examine half of the elements of the array. Another approach is to halve the array, pick the element at the midpoint, compare, then repeat with either the lower or upper half. Divide and conquer. When you run out of elements to inspect or find the element, stop.

An automated test for this technique could be:

```
use Test::More;

my @elements = ( 1, 5, 6, 19, 48, 77, 997, 1025, 7777, 8192, 9999 );

ok   elem_exists(     1, @elements ), 'found first element in array';
```

```
    ok   elem_exists(  9999, @elements ), 'found last element in array';
    ok ! elem_exists(   998, @elements ), 'did not find element not in
array';
    ok ! elem_exists(    -1, @elements ), 'did not find element not in
array';
    ok ! elem_exists( 10000, @elements ), 'did not find element not in
array';

    ok   elem_exists(    77, @elements ), 'found midpoint element';
    ok   elem_exists(    48, @elements ), 'found end of lower half
element';
    ok   elem_exists(   997, @elements ), 'found start of upper half
element';

    done_testing();
```

Recursion is a deceptively simple concept. Every call to a function in Perl creates a new *call frame*, an data structure internal to Perl itself which represents the fact that you've called a function. This call frame includes the lexical environment of the function's current invocation--the values of all lexical variables within the function as invoked. Because the storage of the values of the lexical variables is separate from the function itself, you can have multiple calls to a function active at the same time. A function can even call itself, or *recur*.

To make the previous test pass, write the recursive function `elem_exists()`:

```
    sub elem_exists {
        my ($item, @array) = @_;

        # break recursion with no elements to search
        return unless @array;

        # bias down with odd number of elements
        my $midpoint = int( (@array / 2) - 0.5 );
        my $miditem  = $array[ $midpoint ];

        # return true if found
        return 1 if $item  == $miditem;

        # return false with only one element
        return   if @array == 1;

        # split the array down and recurse
        return B<elem_exists>(
            $item, @array[0 .. $midpoint]
        ) if $item < $miditem;

        # split the array and recurse
        return B<elem_exists>(
             $item, @array[ $midpoint + 1 .. $#array ]
        );
    }
```

Keep in mind that the arguments to the function will be *different* for every call, otherwise the function would always behave the same way (it would continue recursing until the program crashes). That's why the termination condition is so important.

Every recursive program can be written without recursion, but this divide-and-conquer approach is an effective way to manage many similar types of problems. For more information about recursion, iteration, and advanced function use in Perl the free book *Higher Order Perl*http://hop.perl.plover.com/ is an excellent reference.

Lexicals

Every invocation of a function creates its own *instance* of a lexical scope represented internally by a call frame. Even though the declaration of `elem_exists()` creates a single scope for the lexicals `$item`, `@array`, `$midpoint`, and `$miditem`, every *call* to `elem_exists()`--even recursively--stores the values of those lexicals separately.

Not only can `elem_exists()` call itself, but the lexical variables of each invocation are safe and separate:

```
B<use Carp 'cluck';>

sub elem_exists {
    my ($item, @array) = @_;

    B<cluck "[$item] (@array)";>
    ...
}
```

Tail Calls

One *drawback* of recursion is that it's easy to write a function which calls itself infinitely. `elem_exists()` function has several `return` statements for this reason. Perl offers a helpful `Deep recursion on subroutine` warning when it suspects runaway recursion. The limit of 100 recursive calls is arbitrary, but often useful. Disable this warning with `no warnings 'recursion'`.

Because each call to a function requires a new call frame and lexical storage space, highly-recursive code can use more memory than iterative code. *Tail call elimination* can help.

A *tail call* is a call to a function which directly returns that function's results. These recursive calls to `elem_exists()`:

```
# split the array down and recurse
return B<elem_exists>(
    $item, @array[0 .. $midpoint]
) if $item < $miditem;

# split the array and recurse
return B<elem_exists>(
     $item, @array[ $midpoint + 1 .. $#array ]
);
```

... are candidates for tail call elimination. This optimization would avoid returning to the current call and then returning to the parent call. Instead, it returns to the parent call directly.

Perl does not eliminate tail calls automatically, but you can get the same effect by using a special form of the `goto` builtin. Unlike the form which often produces spaghetti code, the `goto` function form replaces the current function call with a call to another function. You may use a function by name or by reference. Manipulate `@_` to modify the arguments passed to the replacement function:

```
# split the array down and recurse
```

```
    if ($item < $miditem) {
        @_ = ($item, @array[0 .. $midpoint]);
        B<goto &elem_exists;>
    }

    # split the array up and recurse
    else {
        @_ = ($item, @array[$midpoint + 1 .. $#array] );
        B<goto &elem_exists;>
    }
```

Sometimes optimizations are ugly, but if the alternative is highly recursive code which runs out of memory, embrace the ugly and rejoice in the practical.

Pitfalls and Misfeatures

Perl still supports old-style invocations of functions, carried over from ancient versions of Perl. Previous versions of Perl required you to invoke functions with a leading ampersand (&) character:

```
    # outdated style; avoid
    my $result = B<&>calculate_result( 52 );

    # very outdated; truly avoid
    my $result = B<do &>calculate_result( 42 );
```

While the vestigial syntax is visual clutter, the leading ampersand form has other surprising behaviors. First, it disables any prototype checking. Second, it *implicitly* passes the contents of `@_` unmodified, unless you've explicitly passed arguments yourself. That unfortunate behavior can be confusing invisible action at a distance.

A final pitfall comes from leaving the parentheses off of function calls. The Perl parser uses several heuristics to resolve ambiguous barewords and the number of parameters passed to a function. Heuristics can be wrong:

```
    # warning; contains a subtle bug
    ok elem_exists 1, @elements, 'found first element';
```

The call to `elem_exists()` will gobble up the test description intended as the second argument to `ok()`. Because `elem_exists()` uses a slurpy second parameter, this may go unnoticed until Perl produces warnings about comparing a non-number (the test description, which it cannot convert into a number) with the element in the array.

While extraneous parentheses can hamper readability, thoughtful use of parentheses can clarify code to readers and to Perl itself.

Scope

Everything with a name in Perl (a variable, a function, a filehandle, a class) has a scope. This *scope* governs the lifespan and visibility of these entities. Scoping helps to enforce *encapsulation*--keeping related concepts together and preventing their details from leaking.

Lexical Scope

Lexical scope is the scope apparent to the readers of a program. Any block delimited by curly braces creates a new scope: a bare block, the block of a loop construct, the block of a `sub` declaration, an `eval` block, a `package` block, or any other non-quoting block. The Perl compiler resolves this scope during compilation.

Lexical scope describes the visibility of variables declared with `my`--*lexical* variables. A lexical variable declared in one scope is visible in that scope and any scopes nested within it, but is invisible to sibling or outer scopes:

```
# outer lexical scope
{
    package Robot::Butler

    # inner lexical scope
    my $battery_level;

    sub tidy_room {
        # further inner lexical scope
        my $timer;

        do {
            # innermost lexical scope
            my $dustpan;
            ...
        } while (@_);

        # sibling inner lexical scope
        for (@_) {
            # separate innermost scope
            my $polish_cloth;
            ...
        }
    }
}
```

... `$battery_level` is visible in all four scopes. `$timer` is visible in the method, the `do` block, and the `for` loop. `$dustpan` is visible only in the `do` block and `$polish_cloth` within the `for` loop.

Declaring a lexical in an inner scope with the same name as a lexical in an outer scope hides, or *shadows*, the outer lexical within the inner scope. For example:

```
my $name = 'Jacob';

{
    my $name = 'Edward';
    say $name;
}

say $name;
```

The silly lexical shadowing example program prints `Edward` and then `Jacob` (don't worry; they're family members, not vampires) because the lexical in the nested scope hides the lexical in the outer scope. Shadowing a lexical is a feature of encapsulation. Declaring multiple variables with the same name and type *in the same lexical scope* produces a warning message.

In real code with larger scopes, this shadowing behavior is often desirable--it's easier to understand code when a lexical is in scope only for a couple of dozen lines. Lexical shadowing *can* happen by accident, though. Limit the scope of variables and the nesting of scopes to lessen your risk.

Some lexical declarations have subtleties, such as a lexical variable used as the iterator variable of a `for` loop. Its declaration occurs *outside* of the block, but its scope is that *within* the loop block:

```
my $cat = 'Brad';

for my $cat (qw( Jack Daisy Petunia Tuxedo Choco )) {
    say "Inner cat is $cat";
}

say "Outer cat is $cat";
```

Functions--named and anonymous--provide lexical scoping to their bodies. This enables closures (*closures*).

Our Scope

Within a scope you may declare an alias to a package variable with the `our` builtin. Like `my`, `our` enforces lexical scoping of the alias. The fully-qualified name is available everywhere, but the lexical alias is visible only within its scope.

`our` is most useful with package global variables such as `$VERSION` and `$AUTOLOAD`. You get a little bit of typo detection (declaring a package global with `our` satisfies the `strict` pragma's `vars` rule), but you still have to deal with a global variable.

Dynamic Scope

Dynamic scope resembles lexical scope in its visibility rules, but instead of looking outward in compile-time scopes, lookup traverses backwards through all of the function calls you've made to reach the current code. Dynamic scope applies only to global and package global variables (as lexicals aren't visible outside their scopes). While a package global variable may be *visible* within all scopes, its *value* may change depending on `local`ization and assignment:

```
our $scope;

sub inner {
    say $scope;
}

sub main {
    say $scope;
    local $scope = 'main() scope';
    middle();
}

sub middle {
    say $scope;
    inner();
}
```

```
    $scope = 'outer scope';
    main();
    say $scope;
```

The program begins by declaring an `our` variable, `$scope`, as well as three functions. It ends by assigning to `$scope` and calling `main()`.

Within `main()`, the program prints `$scope`'s current value, `outer scope`, then `local`izes the variable. This changes the visibility of the symbol within the current lexical scope *as well as* in any functions called from the *current* lexical scope; that *as well as* condition is what dynamic scoping does. Thus, `$scope` contains `main() scope` within the body of both `middle()` and `inner()`. After `main()` returns, when control flow reaches the end of its block, Perl restores the original value of the `local`ized `$scope`. The final `say` prints `outer scope` once again.

Perl also uses different storage mechanisms for package variables and lexical variables. Every scope which contains lexical variables uses a data structure called a *lexical pad* or *lexpad* to store the values for its enclosed lexical variables. Every time control flow enters one of these scopes, Perl creates another lexpad to contain the values of the lexical variables for that particular call. This makes functions work correctly, especially recursive functions (*recursion*).

Each package has a single *symbol table* which holds package variables and well as named functions. Importing (*importing*) works by inspecting and manipulating this symbol table. So does `local`. This is why you may only `local`ize global and package global variables--never lexical variables.

`local` is most often useful with magic variables. For example, `$/`, the input record separator, governs how much data a `readline` operation will read from a filehandle. `$!`, the system error variable, contains error details for the most recent system call. `$@`, the Perl `eval` error variable, contains any error from the most recent `eval` operation. `$|`, the autoflush variable, governs whether Perl will flush the currently `select`ed filehandle after every write operation.

`local`izing these in the narrowest possible scope limits the effect of your changes. This can prevent strange behavior in other parts of your code.

State Scope

Perl's `state` keyword allows you to declare a lexical which has a one-time initialization as well as value persistence:

```
    sub counter {
        B<state> $count = 1;
        return $count++;
    }

    say counter();
    say counter();
    say counter();
```

On the first call to `counter`, Perl initializes `$count`. On subsequent calls, `$count` retains its previous value. This program prints `1`, `2`, and `3`. Change `state` to `my` and the program will print `1`, `1`, and `1`.

You may use an expression to set a `state` variable's initial value:

```
sub counter {
    state $count = shift;
    return $count++;
}

say counter(B<2>);
say counter(B<4>);
say counter(B<6>);
```

Even though a simple reading of the code may suggest that the output should be 2, 4, and 6, the output is actually 2, 3, and 4. The first call to the sub `counter` sets the `$count` variable. Subsequent calls will not change its value.

`state` can be useful for establishing a default value or preparing a cache, but be sure to understand its initialization behavior if you use it:

```
sub counter {
    state $count = shift;
    say 'Second arg is: ', shift;
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

The counter for this program prints 2, 3, and 4 as expected, but the values of the intended second arguments to the `counter()` calls are `two`, 4, and 6--because the `shift` of the first argument only happens in the first call to `counter()`. Either change the API to prevent this mistake, or guard against it with:

```
sub counter {
    my ($initial_value, $text) = @_;

    state $count = $initial_value;
    say "Second arg is: $text";
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

Anonymous Functions

An *anonymous function* is a function without a name. It behaves exactly like a named function--you can invoke it, pass it arguments, return values from it, and take references to it. Yet you can only access an anonymous function by reference (*function_references*), not by name.

A Perl idiom known as a *dispatch table* uses hashes to associate input with behavior:

```
my %dispatch = (
    plus     => \&add_two_numbers,
    minus    => \&subtract_two_numbers,
    times    => \&multiply_two_numbers,
);

sub add_two_numbers      { $_[0] + $_[1] }
sub subtract_two_numbers { $_[0] - $_[1] }
sub multiply_two_numbers { $_[0] * $_[1] }

sub dispatch {
    my ($left, $op, $right) = @_;

    return unless exists $dispatch{ $op };

    return $dispatch{ $op }->( $left, $right );
}
```

The `dispatch()` function takes arguments of the form `(2, 'times', 2)`, evaluates the operation, and returns the result. A trivial calculator application could use `dispatch` to figure out which calculation to perform based on user input.

Declaring Anonymous Functions

The `sub` builtin used without a name creates and returns an anonymous function. Use this function reference where you'd use a reference to a named function, such as to declare the dispatch table's functions in place:

```
    my %dispatch = (
        plus      => sub { $_[0]  + $_[1] },
        minus     => sub { $_[0]  - $_[1] },
        times     => sub { $_[0]  * $_[1] },
        dividedby => sub { $_[0]  / $_[1] },
        raisedto  => sub { $_[0] ** $_[1] },
    );
```

Only those functions within this dispatch table are available for users to call. If your dispatch function used a user-provided string as the literal name of functions, a malicious user could call any function anywhere by passing a fully-qualified name such as `'Internal::Functions::malicious_function'`.

You may also see anonymous functions passed as function arguments:

```
    sub invoke_anon_function {
        my $func = shift;
        return $func->( @_ );
    }

    sub named_func {
        say 'I am a named function!';
    }

    invoke_anon_function( \&named_func );
    invoke_anon_function( B<sub { say 'Who am I?' }> );
```

Anonymous Function Names

Use introspection to determine whether a function is named or anonymous, whether through `caller()` or the CPAN module `Sub::Identify`'s `sub_name()` function:

```
package ShowCaller;

sub show_caller {
    my ($package, $file, $line, $sub) = caller(1);
    say "Called from $sub in $package:$file:$line";
}

sub main {
    my $anon_sub = sub { show_caller() };
    show_caller();
    $anon_sub->();
}

main();
```

The result may be surprising:

```
Called from ShowCaller::B<main>
         in ShowCaller:anoncaller.pl:20
Called from ShowCaller::B<__ANON__>
         in ShowCaller:anoncaller.pl:17
```

The `__ANON__` in the second line of output demonstrates that the anonymous function has no name that Perl can identify. The CPAN module `Sub::Name`'s `subname()` function allows you to attach names to anonymous functions:

```
use Sub::Name;
use Sub::Identify 'sub_name';

my $anon  = sub {};
say sub_name( $anon );

my $named = subname( 'pseudo-anonymous', $anon );
say sub_name( $named );
say sub_name( $anon );

say sub_name( sub {} );
```

This program produces:

```
__ANON__
pseudo-anonymous
pseudo-anonymous
__ANON__
```

Be aware that both references refer to the same underlying anonymous function. Using `subname()` on one reference will change that underlying function; all other references to that function will see the new name.

Implicit Anonymous Functions

Perl allows you to declare anonymous functions as function arguments without using the `sub` keyword. Though this feature exists nominally to enable programmers to write their own syntax such as that for `map` and `eval` (*prototypes*), you can use it for other things, such as to write *delayed* functions that don't look like functions.

Consider the CPAN module `Test::Fatal`, which takes an anonymous function as the first argument to its `exception()` function:

```
use Test::More;
use Test::Fatal;

my $croaker = exception { die 'I croak!' };
my $liver   = exception { 1 + 1 };

like $croaker, qr/I croak/, 'die() should croak';
is   $liver,   undef,       'addition should live';

done_testing();
```

You might rewrite this more verbosely as:

```
my $croaker = exception( sub { die 'I croak!' } );
my $liver   = exception( sub { 1 + 1 } );
```

... or to pass named functions by reference:

```
B<sub croaker { die 'I croak!' }>
B<sub liver   { 1 + 1 }>

my $croaker = exception \&croaker;
my $liver   = exception \&liver;

like $croaker, qr/I croak/, 'die() should die';
is   $liver,   undef,       'addition should live';
```

... but you may *not* pass them as scalar references:

```
B<my $croak_ref = \&croaker;>
B<my $live_ref  = \&liver;>

# BUGGY: does not work
my $croaker   = exception $croak_ref;
my $liver     = exception $live_ref;
```

... because the prototype changes the way the Perl parser interprets this code. It cannot determine with 100% clarity *what* `$croaker` and `$liver` will contain, and so will throw an exception.

```
Type of arg 1 to Test::Fatal::exception
   must be block or sub {} (not private variable)
```

A function which takes an anonymous function as the first of multiple arguments *cannot* have a trailing comma after the function block:

```
use Test::More;
use Test::Fatal 'dies_ok';

dies_ok { die 'This is my boomstick!' } 'No movie references here';
```

This is an occasionally confusing wart on otherwise helpful syntax, courtesy of a quirk of the Perl parser. The syntactic clarity available by promoting bare blocks to anonymous functions can be helpful, but use it sparingly and document the API with care.

Closures

The computer science term *higher order functions* refers to functions which manipulate other functions. Every time control flow enters a function, that function gets a new environment representing that invocation's lexical scope (*scope*). That applies equally well to anonymous functions (*anonymous_functions*). The implication is powerful, and closures show off this power.

Creating Closures

A *closure* is a function that uses lexical variables from an outer scope. You've probably already created and used closures without realizing it:

```
use Modern::Perl '2015';

my $filename = shift @ARGV;

sub get_filename { return $filename }
```

If this code seems straightforward to you, good! *Of course* the `get_filename()` function can see the `$filename` lexical. That's how scope works!

Suppose you want to iterate over a list of items without managing the iterator yourself. You can create a function which returns a function that, when invoked, will return the next item in the iteration:

```
sub make_iterator {
    my @items = @_;
    my $count = 0;

    return sub {
        return if $count == @items;
        return $items[ $count++ ];
    }
}

my $cousins = make_iterator(qw(
    Rick Alex Kaycee Eric Corey Mandy Christine Alex
));

say $cousins->() for 1 .. 6;
```

Even though `make_iterator()` has returned, the anonymous function stored in `$cousins` has closed over the values of these variables *as they existed within* the invocation of `make_iterator()` --and their values persist (*reference_counts*).

Because invoking `make_iterator()` creates a separate lexical environment, the anonymous sub it creates and returns closes over a unique lexical environment for each invocation:

```
my $aunts = make_iterator(qw(
    Carole Phyllis Wendy Sylvia Monica Lupe
));

say $cousins->();
say $aunts->();
```

Because `make_iterator()` does not return these lexicals by value or by reference, only the closure can access them. They're encapsulated as effectively as any other lexical is, although any code which shares a lexical environment can access these values. This idiom provides better encapsulation of what would otherwise be a file or package global variable:

```
{
    my $private_variable;

    sub set_private { $private_variable = shift }
    sub get_private { $private_variable }
}
```

Named functions have package global scope, thus you cannot *nest* named functions. Any lexical variables shared between nested functions will go unshared when the outer function destroys its first lexical environment. Perl will warn you when this happens.

The CPAN module `PadWalker` lets you violate lexical encapsulation, but anyone who uses it gets to fix any bugs that result.

Uses of Closures

Iterating over a fixed-sized list with a closure is interesting, but closures can do much more, such as iterating over a list which is too expensive to calculate or too large to maintain in memory all at once. Consider a function to create the Fibonacci series as you need its elements (probably so you can check the output of your Haskell homework). Instead of recalculating the series recursively, use a cache and lazily create the elements you need:

```
sub gen_fib {
    my @fibs = (0, 1);

    return sub {
        my $item = shift;

        if ($item >= @fibs) {
            for my $calc (@fibs .. $item) {
                $fibs[$calc] = $fibs[$calc - 2]
                             + $fibs[$calc - 1];
            }
        }
        return $fibs[$item];
    }
}

# calculate 42nd Fibonacci number
my $fib = gen_fib();
say $fib->( 42 );
```

Every call to the function returned by `gen_fib()` takes one argument, the *n*th element of the Fibonacci series. The function generates and caches all preceding values in the series as necessary, and returns the requested element.

Here's where closures and first class functions get interesting. This code does two things; there's a pattern specific to caching intertwined with the numeric series. What happens if you extract the cache-specific code (initialize a cache, execute custom code to populate cache elements, and return the calculated or cached value) to a function `gen_caching_closure()`?

```
sub gen_caching_closure {
    my ($calc_element, @cache) = @_;

    return sub {
        my $item = shift;

        $calc_element->($item, \@cache) unless $item < @cache;

        return $cache[$item];
    };
}

sub gen_fib {
    my @fibs = (0, 1, 1);

    return B<gen_caching_closure>( sub {
            my ($item, $fibs) = @_;

            for my $calc ((@$fibs - 1) .. $item) {
                $fibs->[$calc] = $fibs->[$calc - 2]
                               + $fibs->[$calc - 1];
            }
        }B<, @fibs>
    );
}
```

The program behaves as it did before, but now function references and closures separate the cache initialization behavior from the calculation of the next number in the Fibonacci series. Customizing the behavior of code--in this case, `gen_caching_closure()`--by passing in a function allows tremendous flexibility and can clean up your code.

The builtins `map`, `grep`, and `sort` are themselves higher-order functions.

Closures and Partial Application

Closures can also *remove* unwanted genericity. Consider the case of a function which takes several parameters:

```
sub make_sundae {
    my %args      = @_;

    my $ice_cream = get_ice_cream( $args{ice_cream} );
    my $banana    = get_banana(    $args{banana}    );
    my $syrup     = get_syrup(     $args{syrup}     );
    ...
}
```

Myriad customization possibilities might work very well in a full-sized ice cream store, but for an ice cream cart where you only serve French vanilla ice cream on Cavendish bananas, every call to `make_sundae()` passes arguments that never change.

Partial application allows you to bind *some* of the arguments to a function now so that you can provide the others later. Wrap the function you intend to call in a closure and pass the bound arguments. For your ice cream cart:

```
my $make_cart_sundae = sub {
    return make_sundae( @_,
        ice_cream => 'French Vanilla',
        banana    => 'Cavendish',
    );
};
```

Now whenever you process an order, invoke the function reference in `$make_cart_sundae` and pass only the interesting arguments. You'll never forget the invariants or pass them incorrectly. You can even use `Sub::Install` from the CPAN to import `$make_cart_sundae` function into another namespace.

This is only the start of what you can do with higher order functions. Mark Jason Dominus's *Higher Order Perl* is the canonical reference on first-class functions and closures in Perl. Read it online at http://hop.perl.plover.com/.

State versus Closures

Closures (*closures*) use lexical scope (*scope*) to control access to lexical variables--even with named functions:

```
{
    my $safety = 0;

    sub enable_safety  { $safety = 1 }
    sub disable_safety { $safety = 0 }

    sub do_something_awesome {
        return if $safety;
        ...
    }
}
```

All three functions encapsulate that shared state without exposing the lexical variable outside of their shared scope. This idiom works well for cases where multiple functions access that lexical, but it's clunky when only one function does. Suppose every hundredth ice cream parlor customer gets free sprinkles:

```
my $cust_count = 0;

sub serve_customer {
    $cust_count++;
    my $order = shift;
```

```
        add_sprinkles($order) if $cust_count % 100 == 0;
        ...
    }
```

This approach *works*, but creating a new outer lexical scope for a single function is a little bit noisy. The `state` builtin allows you to declare a lexically scoped variable with a value that persists between invocations:

```
    sub serve_customer {
        B<state $cust_count = 0;>
        $cust_count++;

        my $order = shift;
        add_sprinkles($order)
            if ($cust_count % 100 == 0);

        ...
    }
```

`state` also works within anonymous functions:

```
    sub make_counter {
        return sub {
             B<state $count = 0;>
             return $count++;
         }
    }
```

... though there are few obvious benefits to this approach.

State versus Pseudo-State

In old versions of Perl, a named function could close over its previous lexical scope by abusing a quirk of implementation. Using a postfix conditional which evaluates to false with a `my` declaration avoided *reinitializing* a lexical variable to `undef` or its initialized value.

Now any use of a postfix conditional expression modifying a lexical variable declaration produces a deprecation warning. It's too easy to write inadvertently buggy code with this technique; use `state` instead where available, or a true closure otherwise. Rewrite this idiom when you encounter it:

```
    sub inadvertent_state {
        # my $counter  = 1 if 0; # DEPRECATED; don't use
        state $counter = 1;      # prefer

        ...
    }
```

You may only initialize a state variable with a scalar value. If you need to keep track of an aggregate, use a hash or array reference (*references*).

Attributes

Named entities in Perl--variables and functions--can have additional metadata attached in the form of *attributes*. These attributes are arbitrary names and values used with certain types of metaprogramming (*code_generation*).

Attribute declaration syntax is awkward, and using attributes effectively is more art than science. Most programs never use them, but when used well they offer clarity and maintenance benefits.

A simple attribute is a colon-preceded identifier attached to a declaration:

```
    my $fortress      B<:hidden>;

    sub erupt_volcano B<:ScienceProject> { ... }
```

When Perl parses these declarations, it invokes attribute handlers named `hidden` and `ScienceProject`, if they exist for the appropriate types (scalars and functions, respectively). These handlers can do *anything*. If the appropriate handlers do not exist, Perl will throw a compile-time exception.

Attributes may include a list of parameters. Perl treats these parameters as lists of constant strings. The `Test::Class` module from the CPAN uses such parametric arguments to good effect:

```
    sub setup_tests          :Test(setup)    { ... }
    sub test_monkey_creation :Test(10)       { ... }
    sub shutdown_tests       :Test(teardown) { ... }
```

The `Test` attribute identifies methods which include test assertions and optionally identifies the number of assertions the method intends to run. While introspection (*reflection*) of these classes could discover the appropriate test methods, given well-designed solid heuristics, the `:Test` attribute is unambiguous. `Test::Class` provides attribute handlers which keep track of these methods. When the class has finished parsing, `Test::Class` can loop through the list of test methods and run them.

The `setup` and `teardown` parameters allow test classes to define their own support methods without worrying about conflicts with other such methods in other classes. This separates the idea of what this class must do from how other classes do their work. Otherwise a test class might have only one method named `setup` and one named `teardown` and would have to do everything there, then call the parent methods, and so on.

Drawbacks of Attributes

Attributes have their drawbacks. The canonical pragma for working with attributes (the `attributes` pragma) has listed its interface as experimental for many years, and for good reason. Damian Conway's core module `Attribute::Handlers` is much easier to use, and Andrew Main's `Attribute::Lexical` is a newer approach. Prefer either to `attributes` whenever possible.

The worst feature of attributes is that they make it easy to warp the syntax of Perl in unpredictable ways. You may not be able to predict what code with attributes will do. Good documentation helps, but if an innocent-looking declaration on a lexical variable stores a reference to that variable somewhere, your expectations of its lifespan may be wrong. Likewise, a handler may wrap a function in another function and replace it in the symbol table without your knowledge--consider a `:memoize` attribute which automatically invokes the core `Memoize` module.

Attributes *can* help you to solve difficult problems or to make an API much easier to use. When used properly, they're powerful--but most programs never need them.

AUTOLOAD

Perl does not require you to declare every function before you call it. Perl will happily attempt to call a function even if it doesn't exist. Consider the program:

```
use Modern::Perl;

bake_pie( filling => 'apple' );
```

When you run it, Perl will throw an exception due to the call to the undefined function `bake_pie()`.

Now add a function called `AUTOLOAD()`:

```
sub AUTOLOAD {}
```

When you run the program now, nothing obvious will happen. Perl will call a function named `AUTOLOAD()` in a package--if it exists--whenever normal dispatch fails. Change the `AUTOLOAD()` to emit a message to demonstrate that it gets called:

```
sub AUTOLOAD { B<say 'In AUTOLOAD()!'> }
```

The `AUTOLOAD()` function receives the arguments passed to the undefined function in `@_` and the fully-qualified *name* of the undefined function in the package global `$AUTOLOAD` (here, `main::bake_pie`):

```
sub AUTOLOAD {
    B<our $AUTOLOAD;>

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) B<for $AUTOLOAD>!"
}
```

Extract the method name with a regular expression (*chp.regex*):

```
sub AUTOLOAD {
    B<my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) B<for $name>!"
}
```

Whatever `AUTOLOAD()` returns, the original call receives:

```
say secret_tangent( -1 );

sub AUTOLOAD { return 'mu' }
```

So far, these examples have merely intercepted calls to undefined functions. You have other options.

Redispatching Methods in AUTOLOAD()

A common pattern in OO programming (*moose*) is to *delegate* or *proxy* certain methods from one object to another. A logging proxy can help with debugging:

```
package Proxy::Log;

# constructor blesses reference to a scalar
```

```
sub AUTOLOAD {
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    Log::method_call( $name, @_ );

    my $self = shift;
    return $$self->$name( @_ );
}
```

This `AUTOLOAD()` extracts the name of the undefined method. Then it dereferences the proxied object from a blessed scalar reference, logs the method call, then invokes that method on the proxied object with the provided parameters.

Generating Code in AUTOLOAD()

This double dispatch is easy to write but inefficient. Every method call on the proxy must fail normal dispatch to end up in `AUTOLOAD()`. Pay that penalty only once by installing new methods into the proxy class as the program needs them:

```
sub AUTOLOAD {
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    my $method = sub { ... };

    B<no strict 'refs';>
    B<*{ $AUTOLOAD } = $method;>
    return $method->( @_ );
}
```

The body of the previous `AUTOLOAD()` has become a closure (*closures*) bound over the *name* of the undefined method. Installing that closure in the appropriate symbol table allows all subsequent dispatch to that method to find the created closure (and avoid `AUTOLOAD()`). This code finally invokes the method directly and returns the result.

Though this approach is cleaner and almost always more transparent than handling the behavior directly in `AUTOLOAD()`, the code *called* by `AUTOLOAD()` may see `AUTOLOAD()` in its `caller()` list. While it may violate encapsulation to care that this occurs, leaking the details of *how* an object provides a method may also violate encapsulation.

Some code uses a tailcall (*tailcalls*) to *replace* the current invocation of `AUTOLOAD()` with a call to the destination method:

```
sub AUTOLOAD {
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    my $method = sub { ... }

    no strict 'refs';
    *{ $AUTOLOAD } = $method;
    B<goto &$method;>
}
```

This has the same effect as invoking `$method` directly, except that `AUTOLOAD()` will no longer appear in the list of calls available from `caller()`, so it looks like normal method dispatch occurred.

Drawbacks of AUTOLOAD

AUTOLOAD() can be useful, though it is difficult to use properly. The naïve approach to generating methods at runtime means that the can() method will not report the right information about the capabilities of objects and classes. The easiest solution is to predeclare all functions you plan to AUTOLOAD() with the subs pragma:

```
    use subs qw( red green blue ochre teal );
```

Forward declarations are useful only in the two rare cases of attributes (*attributes*) and autoloading (*autoload*).

That technique documents your intent well, but requires you to maintain a static list of functions or methods. Overriding can() (*universal*) sometimes works better:

```
    sub can {
        my ($self, $method) = @_;

        # use results of parent can()
        my $meth_ref = $self->SUPER::can( $method );
        return $meth_ref if $meth_ref;

        # add some filter here
        return unless $self->should_generate( $method );

        $meth_ref = sub { ... };
        no strict 'refs';
        return *{ $method } = $meth_ref;
    }

    sub AUTOLOAD {
        my ($self) = @_;
        my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

        return unless my $meth_ref = $self->can( $name );
        goto &$meth_ref;
    }
```

AUTOLOAD() is a big hammer; it can catch functions and methods you had no intention of autoloading, such as DESTROY(), the destructor of objects. If you write a DESTROY() method with no implementation, Perl will happily dispatch to it instead of AUTOLOAD():

```
    # skip AUTOLOAD()
    sub DESTROY {}
```

The special methods import(), unimport(), and VERSION() never go through AUTOLOAD().

If you mix functions and methods in a single namespace which inherits from another package which provides its own AUTOLOAD(), you may see the strange error:

```
  Use of inherited AUTOLOAD for non-method I<slam_door>() is deprecated
```

If this happens to you, simplify your code; you've called a function which does not exist in a package which inherits from a class which contains its own AUTOLOAD(). The problem compounds in several ways: mixing functions and methods in a single namespace is often a design flaw, inheritance and AUTOLOAD() get complex very quickly, and reasoning about code when you don't know what methods objects provide is difficult.

AUTOLOAD() is useful for quick and dirty programming, but robust code avoids it.