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