

## NAME

`carp` - warn of errors (from perspective of caller)

`cluck` - warn of errors with stack backtrace (not exported by default)

`croak` - die of errors (from perspective of caller)

`confess` - die of errors with stack backtrace

## SYNOPSIS

```
use Carp;
croak "We're outta here!";

use Carp qw(cluck);
cluck "This is how we got here!";
```

## DESCRIPTION

The Carp routines are useful in your own modules because they act like `die()` or `warn()`, but with a message which is more likely to be useful to a user of your module. In the case of `cluck`, `confess`, and `longmess` that context is a summary of every call in the call-stack. For a shorter message you can use `carp` or `croak` which report the error as being from where your module was called. There is no guarantee that that is where the error was, but it is a good educated guess.

You can also alter the way the output and logic of Carp works, by changing some global variables in the Carp namespace. See the section on `GLOBAL VARIABLES` below.

Here is a more complete description of how `carp` and `croak` work. What they do is search the call-stack for a function call stack where they have not been told that there shouldn't be an error. If every call is marked safe, they give up and give a full stack backtrace instead. In other words they presume that the first likely looking potential suspect is guilty. Their rules for telling whether a call shouldn't generate errors work as follows:

1. Any call from a package to itself is safe.
2. Packages claim that there won't be errors on calls to or from packages explicitly marked as safe by inclusion in `@CARP_NOT`, or (if that array is empty) `@ISA`. The ability to override what `@ISA` says is new in 5.8.
3. The trust in item 2 is transitive. If A trusts B, and B trusts C, then A trusts C. So if you do not override `@ISA` with `@CARP_NOT`, then this trust relationship is identical to, "inherits from".
4. Any call from an internal Perl module is safe. (Nothing keeps user modules from marking themselves as internal to Perl, but this practice is discouraged.)
5. Any call to Perl's warning system (eg Carp itself) is safe. (This rule is what keeps it from reporting the error at the point where you call `carp` or `croak`.)
6. `$Carp::CarpLevel` can be set to skip a fixed number of additional call levels. Using this is not recommended because it is very difficult to get it to behave correctly.

## Forcing a Stack Trace

As a debugging aid, you can force Carp to treat a `croak` as a `confess` and a `carp` as a `cluck` across *all* modules. In other words, force a detailed stack trace to be given. This can be very helpful when trying to understand why, or from where, a warning or error is being generated.

This feature is enabled by 'importing' the non-existent symbol 'verbose'. You would typically enable it by saying

```
perl -MCarp=verbose script.pl
```

or by including the string `-MCarp=verbose` in the `PERL5OPT` environment variable.

Alternately, you can set the global variable `$Carp::Verbose` to true. See the `GLOBAL VARIABLES` section below.

## GLOBAL VARIABLES

### `$Carp::MaxEvalLen`

This variable determines how many characters of a string-eval are to be shown in the output. Use a value of 0 to show all text.

Defaults to 0.

### `$Carp::MaxArgLen`

This variable determines how many characters of each argument to a function to print. Use a value of 0 to show the full length of the argument.

Defaults to 64.

### `$Carp::MaxArgNums`

This variable determines how many arguments to each function to show. Use a value of 0 to show all arguments to a function call.

Defaults to 8.

### `$Carp::Verbose`

This variable makes `carp` and `croak` generate stack backtraces just like `cluck` and `confess`. This is how `use Carp 'verbose'` is implemented internally.

Defaults to 0.

### `@CARP_NOT`

This variable, *in your package*, says which packages are *not* to be considered as the location of an error. The `carp()` and `cluck()` functions will skip over callers when reporting where an error occurred.

NB: This variable must be in the package's symbol table, thus:

```
# These work
our @CARP_NOT; # file scope
use vars qw(@CARP_NOT); # package scope
@My::Package::CARP_NOT = ... ; # explicit package variable

# These don't work
sub xyz { ... @CARP_NOT = ... } # w/o declarations above
my @CARP_NOT; # even at top-level
```

Example of use:

```
package My::Carping::Package;
use Carp;
our @CARP_NOT;
sub bar { .... or _error('Wrong input') }
sub _error {
    # temporary control of where'ness, __PACKAGE__ is implicit
    local @CARP_NOT = qw(My::Friendly::Caller);
    carp(@_)
}
```

This would make `Carp` report the error as coming from a caller not in `My::Carping::Package`, nor from `My::Friendly::Caller`.

Also read the *DESCRIPTION* section above, about how `Carp` decides where the error is reported from.

Use `@CARP_NOT`, instead of `$Carp::CarpLevel`.

Overrides `Carp`'s use of `@ISA`.

### **%Carp::Internal**

This says what packages are internal to Perl. `Carp` will never report an error as being from a line in a package that is internal to Perl. For example:

```
$Carp::Internal{ (__PACKAGE__) }++;  
# time passes...  
sub foo { ... or confess("whatever") };
```

would give a full stack backtrace starting from the first caller outside of `__PACKAGE__`. (Unless that package was also internal to Perl.)

### **%Carp::CarpInternal**

This says which packages are internal to Perl's warning system. For generating a full stack backtrace this is the same as being internal to Perl, the stack backtrace will not start inside packages that are listed in `%Carp::CarpInternal`. But it is slightly different for the summary message generated by `carp` or `croak`. There errors will not be reported on any lines that are calling packages in `%Carp::CarpInternal`.

For example `Carp` itself is listed in `%Carp::CarpInternal`. Therefore the full stack backtrace from `confess` will not start inside of `Carp`, and the short message from calling `croak` is not placed on the line where `croak` was called.

### **\$Carp::CarpLevel**

This variable determines how many additional call frames are to be skipped that would not otherwise be when reporting where an error occurred on a call to one of `Carp`'s functions. It is fairly easy to count these call frames on calls that generate a full stack backtrace. However it is much harder to do this accounting for calls that generate a short message. Usually people skip too many call frames. If they are lucky they skip enough that `Carp` goes all of the way through the call stack, realizes that something is wrong, and then generates a full stack backtrace. If they are unlucky then the error is reported from somewhere misleading very high in the call stack.

Therefore it is best to avoid `$Carp::CarpLevel`. Instead use `@CARP_NOT`, `%Carp::Internal` and `%Carp::CarpInternal`.

Defaults to 0.

### **BUGS**

The `Carp` routines don't handle exception objects currently. If called with a first argument that is a reference, they simply call `die()` or `warn()`, as appropriate.