Dying With Honor

Peter Chines

YAPC::NA 2003

Abstract

Perl offers a wealth of options for terminating a program and providing feedback to users about exceptional conditions. In addition to the core Perl language constructs, the variety of methods available in the Carp and CGI::Carp modules give Perl programmers a lot of flexibility in dealing with abnormal situations. This paper addresses how to use these tools to best advantage and offers examples of when to use each method of terminating a program. Make sure your programs die honorably, providing all the information needed to trace and correct a problem, while avoiding unnecessary clutter.

1 Dying With Honor

One of the most important duties of a program is to effectively communicate to its users. The form and content of that communication will vary based on the circumstances. It is particularly important to communicate well when the program is terminating. The user wants to know: Did it work? if not, What went wrong? and finally, How do I fix it? Carefully constructed error messages concisely communicate all of this information.

These messages are commonly sent to standard error, but may also be written to a log file, or sent via email, pager, or some other mechanism. The choice of mechanism and the precise amount of information necessary varies, depending both on the exact type of error and the program's context, e.g. whether the program is being debugged, in beta testing, run on an interactive command line, as a subprocess run in batch mode, or as a daemon process.

One thing is constant: if a program must fail, it is better to fail as early as possible. That puts the responsibility on the programmer to think ahead. Particularly in the case of a long-running or potentially destructive activity, a program should check that all necessary resources are in place before committing to the process. Also, it is important to distinguish between normal and abnormal execution; when the program fails, it should usually fail loudly.

Error messages are not the only form of feedback from a terminating program. Status codes are also an important channel for information, particularly in an environment like Unix where one program calls another to perform a task. By convention, a normal exit is indicated by a return value of zero, while non-zero value means that something unusual occurred. This convention is embedded in many of the tools commonly used in a command-line environment, such as `make` and the shell shortcut evaluation operators && and ||.

The object of this article to is familiarize readers with the tools available in Perl's core features as well as the additional features in common modules such as Carp and CGI::Carp, and suggest ways that they may be useful. In the end, each programmer must use his or her own judgment to apply these tools to the particulars of a given situation.

2 CORE Language Features

The basic tools for reporting errors and terminating a Perl program are well known and frequently used. But there are some subtleties in how even these common tools work. Everyone can benefit from reading the documentation for these functions. Here is a quick summary, with some suggestions of situations where they may be useful.

2.1 die

The most commonly used mechanism for terminating a program early is the `die` function. It writes a message to STDERR and terminates with a non-zero status code indicating failure[1]. You may be surprised to learn that such a simple function has several pages of documentation. The reason is that our language designers and implementors have made `die` try to "do the right thing" in a variety of different situations. Knowing exactly how `die` works allows you to tailor its functionality to your particular situation. I will only address the highlights here; see `perldoc -f die` for much more information.

If you specify a message that ends with a newline, it will be written exactly as you specify. If you leave out the newline, however, perl will add the filename and line number where the `die` occurred. If you have read a line from a filehandle, perl will also add the name of the last filehandle read and the number of the last line read from that filehandle.

`die` is used to report all kinds of fatal errors. One common use is for reporting errors accessing the file system, as in this common idiom:

```
open FH, $filename
  or die "Can’t open $filename, $!\n";
```

Note the use of `$!` in the string to pass on the specific operating system error to the user, so that she has enough information to fix the problem. For example, the solution for "No such file or directory" is different from "Permission denied." Note also the ending newline. In general, if a file can't be opened, that problem is not related to a particular line of your program, and thus there is no point in cluttering up the error message with this information.

[1]The actual value returned takes into account the values of the $! (operating system error code) and $? (child process status code). If both of these is zero, the status code returned by `die` to the calling process is 255. One implication of this is that it is possible to control the status code returned by setting $! to an arbitrary value before calling `die`.

On the other hand, with some kinds of errors, it is very appropriate to include the program and line number, such as when your program reaches an unexpected or unimplemented part of your code:

```
if ($type == 1) {
    basic_algorithm($data);
}
elsif ($type == 2) {
    complex_algorithm($data);
}
else {
    die "Sorry, there is no code for type $type yet";
}
```

2.2 warn

The `warn` function works much like `die`, except that it does not terminate the program. The same rules apply regarding the newline at the end of the message.

Warnings are typically issued to report unexpected event or invalid data that it is possible to recover from. They can also be used to issue debugging messages. For all but the simplest tasks, it is probably better to use a full-fledged logging system like Log::Log4perl. Here is a typical use of `warn`:

```
while (<>) {
    chomp;
    if (m/^(\d+)\s+(\w+)^/) {
        # process line
    }
    else {
        warn "Invalid data format";
    }
}
```

This code ignores lines that do not have the correct format, emitting a warning that looks like this:

```
Invalid data format at ./wtest line 10, <> line 3
```

2.3 exit

The `exit` function simply terminates a program, without writing any kind of message. Usually, using `exit` is not necessary in a well-structured program, since the program will terminate when it gets to the end of the code and there are no statements left to execute.

`exit` may be called with a numeric argument, which is returned as the process status code[2]. When `exit` is called with no arguments, the status code returned is zero.

[2]Only the lower 8 bits of the integer part of this argument is returned, since the status code consists of just a single byte.

2.4 END blocks

An END block is code that will run when the program is finishing, regardless of how the program is terminated. This makes the END block a perfect place to release locks or other shared resources that your program has acquired. Perl will do its best to execute this code, whether the program terminates as a result of a `die` statement, a normal exit, or upon receiving a signal from the operating system[3]. Any time your code acquires a persistent resource you should use an END block to ensure that it is properly released under all conditions.

3 Carp.pm

The Carp module, one of the modules distributed with the base Perl installation, offers several methods that are particularly useful to module writers. These methods are not just "fancy" versions of `die` and `warn`, and they should not be used interchangeably with them. To access these methods you must `use Carp;` in the package where you want to use them.

3.1 croak

The `croak` method was designed to report errors that arise from methods being called incorrectly or being passed bad arguments. Like `die`, it writes a message to STDERR and terminates the program[4]. However, it reports the error as having occurred at the point where one of the methods in the current package was called from outside the package and its parent classes, if any. This is an important point that bears repeating: `croak` always reports a file and line number, but the place it reports is not the line where `croak` appears; it can be several subroutine calls away, and almost always in a different file. Thus, if `croak` is misused as a synonym for `die`, it will generate misleading information.

`croak` is most commonly used to report required arguments that are missing, or arguments that are not the expected type. Because the message from `croak` is always followed by the file and line number of the calling code, it reads better if you leave out the trailing newline. The line where the method was called may include several method calls, so it is often a good idea to include the name of the routine that is reporting the error in the message.

Consider the following example. In the module code, we use Carp:

```
package MyModule;
use strict;
use Carp;
sub a_public_method {
    my ($self, $arg) = @_;
```

[3]There are some signals that cannot be caught, and will terminate the program immediately, e.g. SIGKILL. In these cases, no cleanup can be performed. This is why terminating a program using this extreme means should be a last resort.

[4]Unlike die, it always returns 255 as the status code.

```
        croak "a_public_method: missing parameter"
          if !$arg;
        $self->_private_method($arg);
        # ... etc.
    }
    # ... more code
    1;
```

In the calling program, the module is used and its method is called:

```
    #!/usr/bin/perl -w
    use MyModule;
    MyModule->a_public_method();
```

When this code is run, the error reported is:

```
    a_public_method: missing parameter at ./test line 3
```

It is good defensive programming practice for each public method to do its own parameter checking, and `croak` is an ideal tool for this.

3.2 carp

The `carp` method prints a warning to STDERR, but does not terminate the program. `carp` is like `croak`, in that it reports the file and line number of the last line outside the current module where a package method was called.

This makes `carp` an ideal way to report deprecated uses of a module. Say that in version 1.0 of your module there was a public method called `foo()`, but in version 2.0 you decide to replace that method with `bar()`. You could just remove the `foo` method, mention this fact in your documentation and leave it at that. But if there is a lot of code that depends on your module, you might decide to simply issue a warning instead, and put off removing the method until later, giving people time to rework their code. By using `carp` to issue these warnings, you can help the programmers who use your module to pinpoint where their code must be modified. In the module, the `foo` method can even redirect the call to the new `bar` method:

```
    sub foo {
        my ($self, @args) = @_;
        carp "foo is deprecated; use bar instead";
        $self->bar(@args);
    }
```

The code continues to work, while the message explains exactly needs to be done:

```
    foo is deprecated; use bar instead at ./driver line 27
```

3.3 confess

Sometimes more information is needed to trace the source of a problem. The Carp module includes a `confess` method, which returns a nicely formatted stack trace, showing each subroutine call that led to the line where `confess` is executed. After the stack trace, the program is terminated.

A stack trace is overkill for most exception reporting. `confess` is most useful in diagnosing what went wrong when your program reaches a place that it should never get to, particularly when that place is deep in your code and there are multiple paths it could take to get there. A `confess` stack trace looks like this:

```
Something impossible happened at MyModule.pm line 23
    MyModule::c_method('MyModule') called at MyModule.pm line 16
    MyModule::b_method('MyModule', 'c') called at MyModule.pm line 11
    MyModule::a_method('MyModule', 'c') called at ./conftest line 5
```

Note that `confess` includes the arguments used for each subroutine invocation.

Importing the verbose symbol when using the Carp module makes all calls to `croak` behave as if `confess` were called, which may be useful in some debugging situations. It also makes all `carp` calls behave like the `cluck` method.

3.4 cluck

`cluck` displays a stack trace, but does not terminate the program. This method is not commonly used, and thus is not exported by default when the Carp module is used. To access it, you must import it explicitly:

```
use Carp qw(cluck);
```

It would appear that this method is primarily useful in debugging situations, where it is not practical to run the debugger interactively.

4 CGI::Carp

The CGI::Carp module is a boon to everyone who writes CGI-based web applications in Perl. While the module includes all of the same methods that are in the Carp package, its most useful and important features have nothing to do with these methods.

One of these features is automatic when you use CGI::Carp. All warnings and fatal error messages, whether they originate from `warn`, `die`, `carp`, `croak`, etc. are prefixed with a timestamp and the name of the program running. This makes the messages, which are sent by default to the web server's error log, much more informative. This alone makes CGI::Carp worth using.

But wait, there's more. By importing special symbols in the use clause, you can turn on additional CGI::Carp features. Importing `fatalsToBrowser` sends all fatal errors to the browser window, in addition to the error log. This can be a great help in developing web applications, returning your meaningful error message to the browser window, rather than an unhelpful “500 Server Error” page. On the other hand, you must be careful with the information that you put in your error messages when this feature is activated. You should take care not to give away important security information, such as the locations of key files, usernames or passwords.

Importing the `warningsToBrowser` method allows you to direct all non-fatal warnings into HTML comments, which are visible in the HTML page source, but not rendered in the browser window. To use `warningsToBrowser`, you must also call this function (with a true value) after the HTTP headers and initial HTML tags have been sent:

```
use CGI;
use CGI::Carp qw(fatalsToBrowser warningsToBrowser);
my $q = CGI->new();
print $q->header();
warningsToBrowser(1);
```

For more information, see the POD documentation for CGI::Carp.

5 Advanced features

Perl also has some less-well-known features that allow you to exercise greater control over when and how your program issues warnings and terminates.

5.1 Exception Handling with eval

An unexpected condition need not always result in the termination of the program. Unfortunately, an error condition cannot always be handled in the routine where it is detected. When it cannot, the best solution is to throw an exception which propagates up the call stack until it can be handled properly. The alternative, returning error codes and testing for them in each subroutine call, has several drawbacks:

- it is less readable, with error handling code mixed in with the normal process flow,
- it is less efficient, because of the extra tests, and
- most importantly, it is more error prone, since you must remember to check for errors after each call.

Perl uses `die` and `eval` to provide a basic exception handling mechanism[5]. Within an `eval`, would-be fatal errors are trapped, so that they cause an immediate exit from the `eval`, with the error message in the variable `$@`. The program does not terminate, but continues with the line immediately following the `eval` statement. Subsequent statements can check the value of `$@` and take action depending on the error that occurred.

Here’s an example:

[5]For more traditional try/catch semantics, take a look at Error.pm and Exception.pm.

```
while (<>) {
    eval {
        deep_nested_subroutine_that_may_die($_);
    };
    if ($@) {
        if ($@ =~ /expected error/) {
            next;
        }
        else {
            die $@;
        }
    }
}
```

Note that the `eval` block is defined with brackets, rather than quotes. This is more efficient, since the evaled code is only compiled once, rather than every time the code is executed.

5.2 Signal Handlers

One of the special predefined variables in Perl is the `%SIG` hash, which allows you to specify routines to handle various signals from the operating system. Perl extends the signal handler metaphor to include the special signals `__DIE__` and `__WARN__` as well. Establishing signal handlers for `die` and `warn` allow you to customize the behavior of your program whenever these routines are called. And since the Carp module uses the core `die` and `warn` routines internally, these handlers apply to the various Carp methods as well.

To set up a signal handler, simply assign a subroutine reference to the hash element. Signal handlers like this have a variety of uses. You could use them to write all fatal errors to a log file as well as to `STDERR`:

```
open LOG, ">>$logfile"
  or die "Can't append to $logfile, $!\n";
$SIG{__DIE__} = sub { print LOG @_; };
```

Note that even though our handler does not explicitly call `die`, it is implicitly called at the end of the handler. You can, however, call `die` explicitly in the handler; it does not result in an endless loop. If we want to change the error message, for example, prefixing each fatal message with the name of the script that is running, we could set up a handler like this:

```
$SIG{__DIE__} = sub { die $0, ": ", @_; };
```

There is no way to prevent the program from terminating within a `die` signal handler.

Signal handlers for warning are different. If you don't explicitly do something with the warning message, nothing happens. One way to silence all[6]warnings is to set up a

[6]If you want to avoid compile-time warnings as well, then you must establish this signal handler in a BEGIN block near the start of your code.

null handler:

```
$SIG{__WARN__} = sub { };
```

Needless to say, this is rarely a good idea. Another possibility is to convert all warnings to fatal errors:

```
$SIG{__WARN__} = sub { die @_; };
```

Only one handler can be active at any given time, so signal handlers should be used sparingly, and probably not at all in modules or library code unless well documented. In particular, signal handlers like these are the mechanism used by CGI::Carp to modify the content and destination of warnings and fatal error messages, and thus you cannot define your own signal handlers for `die` and `warn` at the same time you are using the CGI::Carp module.

The `die` signal handler is called even within an `eval`, at least in current versions of Perl, so if you use both a signal handler and evals in your program, read the documentation carefully to make sure they interact the way you intend.

6 Conclusion

In summary, an honorable program will:

- fail quickly and loudly
- always provide enough information for someone to identify and fix the problem
- avoid providing more information than is needed
- use END blocks where needed
- use Carp in your modules to report misuse of your module by driver code
- use exception handling, either with `die/eval` or a specialized module, rather than propagating error codes through deeply nested subroutine calls

Writing a program that performs correctly under normal conditions is not easy; writing one that also handles unexpected conditions gracefully is even more challenging, but that is our task. With an understanding of your users and the context in which the program will be run and an in-depth knowledge of your tools, not to mention a dedication to craftsmanship and ruthless testing, you can create high-quality programs that meet this challenge.

A Appendixes

A.1 Resources

Nearly all of the information in this article can be gleaned by a careful reading of Perl's extensive online documentation. In particular, use `perldoc -f` to read the man pages for `die`, `warn`, `exit` and `eval`.

Also use `perldoc` to review the documentation for the standard modules Carp and CGI::Carp, and check out CPAN for additional modules mentioned in this article, such as Error, Exception, and Log::Log4Perl.

A.2 Acknowledgements

I'd like to express my appreciation to Dr. Francis Collins, director of the National Human Genome Research Institute and my boss, who makes NHGRI a great place to work. I'd also like to thank my colleagues at NHGRI, past and present, who have helped me become a better programmer, including Gunther Birznieks, Anthony Masiello, Joseph Ryan, Eric Tachibana, Kenneth Trout, Narisu Narisu, Terry Gliedt, Arjun Prasad, John Pearson, Ingeborg Holt, Ben Hsu and Lowell Umayam. Thanks also to all of the people who have contributed to making Perl a great language, and its documentation such a rich resource for the whole community.