PERL - ERROR HANDLING

The execution and the errors always go together. If you are opening a file which does not exist. then if you did not handle this situation properly then your program is considered to be of bad quality.

The program stops if an error occurs. So a proper error handling is used to handle various type of errors, which may occur during a program execution and take appropriate action instead of halting program completely.

You can identify and trap an error in a number of different ways. Its very easy to trap errors in Perl and then handling them properly. Here are few methods which can be used.

The if statement

The **if statement** is the obvious choice when you need to check the return value from a statement; for example −

```
if(open(DATA, $file)){
   ...
}else{
   die "Error: Couldn't open the file - $!";
}
```

Here variable $! returns the actual error message. Alternatively, we can reduce the statement to one line in situations where it makes sense to do so; for example −

```
open(DATA, $file) || die "Error: Couldn't open the file $!";
```

The unless Function

The **unless** function is the logical opposite to if: statements can completely bypass the success status and only be executed if the expression returns false. For example −

```
unless(chdir("/etc")){
   die "Error: Can't change directory - $!";
}
```

The **unless** statement is best used when you want to raise an error or alternative only if the expression fails. The statement also makes sense when used in a single-line statement −

```
die "Error: Can't change directory!: $!" unless(chdir("/etc"));
```

Here we die only if the chdir operation fails, and it reads nicely.

The ternary Operator

For very short tests, you can use the conditional operator **?−**

```
print(exists($hash{value}) ? 'There' : 'Missing',"\n");
```

It's not quite so clear here what we are trying to achieve, but the effect is the same as using an if or unless statement. The conditional operator is best used when you want to quickly return one of the two values within an expression or statement.

The warn Function

The warn function just raises a warning, a message is printed to STDERR, but no further action is taken. So it is more useful if you just want to print a warning for the user and proceed with rest of the operation −

```
chdir('/etc') or warn "Can't change directory";
```

The die Function

The die function works just like warn, except that it also calls exit. Within a normal script, this function has the effect of immediately terminating execution. You should use this function in case it is useless to proceed if there is an error in the program −

```
chdir('/etc') or die "Can't change directory";
```

Errors within Modules

There are two different situations we should be able to handle −

- Reporting an error in a module that quotes the module's filename and line number - this is useful when debugging a module, or when you specifically want to raise a module-related, rather than script-related, error.
- Reporting an error within a module that quotes the caller's information so that you can debug the line within the script that caused the error. Errors raised in this fashion are useful to the end-user, because they highlight the error in relation to the calling script's origination line.

The **warn** and **die** functions work slightly differently than you would expect when called from within a module. For example, the simple module −

```
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function {
   warn "Error in module!";
}
1;
```

When called from a script like below −

```
use T;
function();
```

It will produce the following result:

```
Error in module! at T.pm line 9.
```

This is more or less what you might expected, but not necessarily what you want. From a module programmer's perspective, the information is useful because it helps to point to a bug within the module itself. For an end-user, the information provided is fairly useless, and for all but the hardened programmer, it is completely pointless.

The solution for such problems is the Carp module, which provides a simplified method for reporting errors within modules that return information about the calling script. The Carp module provides four functions: carp, cluck, croak, and confess. These functions are discussed below.

The carp Function

The carp function is the basic equivalent of warn and prints the message to STDERR without actually exiting the script and printing the script name.

```
package T;
```

```
require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function {
   carp "Error in module!";
}
1;
```

When called from a script like below −

```
use T;
function();
```

It will produce the following result −

```
Error in module! at test.pl line 4
```

The cluck Function

The cluck function is a sort of supercharged carp, it follows the same basic principle but also prints a stack trace of all the modules that led to the function being called, including the information on the original script.

```
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp qw(cluck);

sub function {
   cluck "Error in module!";
}
1;
```

When called from a script like below −

```
use T;
function();
```

It will produce the following result −

```
Error in module! at T.pm line 9
 T::function() called at test.pl line 4
```

The croak Function

The **croak** function is equivalent to **die**, except that it reports the caller one level up. Like die, this function also exits the script after reporting the error to STDERR −

```
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function {
   croak "Error in module!";
}
1;
```

When called from a script like below −

```
use T;
function();
```

It will produce the following result −

```
Error in module! at test.pl line 4
```

As with carp, the same basic rules apply regarding the including of line and file information according to the warn and die functions.

The confess Function

The **confess** function is like **cluck**; it calls die and then prints a stack trace all the way up to the origination script.

```
package T;

require Exporter;
@ISA = qw/Exporter/;
@EXPORT = qw/function/;
use Carp;

sub function {
   confess "Error in module!";
}
1;
```

When called from a script like below −

```
use T;
function();
```

It will produce the following result −

```
Error in module! at T.pm line 9
 T::function() called at test.pl line 4
```