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
```

Socket Programming

What is a Socket?

Socket is a Berkeley UNIX mechanism of creating a virtual duplex connection between different processes. This was later ported on to every known OS enabling communication between systems across geographical location running on different OS software. If not for the socket, most of the network communication between systems would never ever have happened.

Taking a closer look; a typical computer system on a network receives and sends information as desired by the various applications running on it. This information is routed to the system, since a unique IP address is designated to it. On the system, this information is given to the relevant applications, which listen on different ports. For example an internet browser listens on port 80 for information received from the web server. Also we can write our custom applications which may listen and send/receive information on a specific port number.

For now, let's sum up that a socket is an IP address and a port, enabling connection to send and recieve data over a network.

To explain above mentioned socket concept we will take an example of Client - Server Programming using Perl. To complete a client server architecture we would have to go through the following steps –

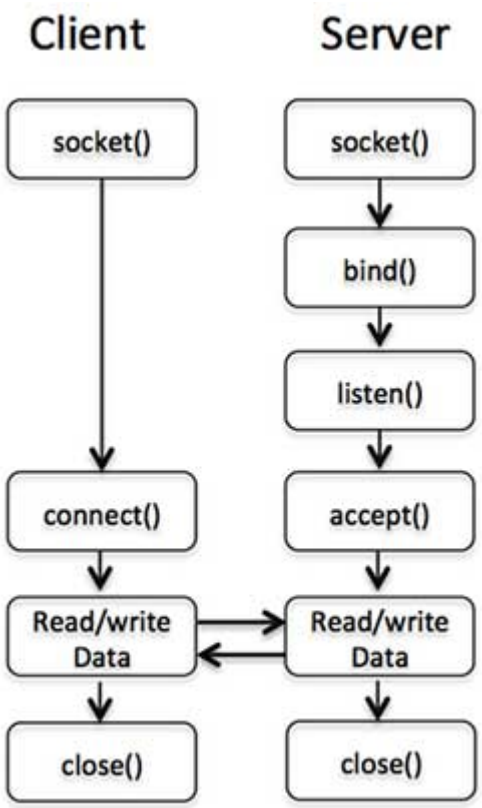
To Create a Server

- Create a socket using **socket** call.
- Bind the socket to a port address using **bind** call.
- Listen to the socket at the port address using **listen** call.
- Accept client connections using **accept** call.

To Create a Client

- Create a socket with **socket** call.
- Connect (the socket) to the server using **connect** call.

Following diagram shows the complete sequence of the calls used by Client and Server to communicate with each other −



Server Side Socket Calls

The socket() call

The **socket()** call is the first call in establishing a network connection is creating a socket. This call has the following syntax −

socket(SOCKET, DOMAIN, TYPE, PROTOCOL);

The above call creates a SOCKET and other three arguments are integers which should have the following values for TCP/IP connections.

- **DOMAIN** should be PF_INET. It's probable 2 on your computer.
- **TYPE** should be SOCK_STREAM for TCP/IP connection.
- **PROTOCOL** should be **(getprotobyname('tcp'))[2]**. It is the particular protocol such as TCP to be spoken over the socket.

So socket function call issued by the server will be something like this −

```
use Socket     # This defines PF_INET and SOCK_STREAM

socket(SOCKET,PF_INET,SOCK_STREAM,(getprotobyname('tcp'))[2]);
```

The bind() call

The sockets created by socket() call are useless until they are bound to a hostname and a port number. Server uses the following **bind()** function to specify the port at which they will be accepting connections from the clients.

```
bind( SOCKET, ADDRESS );
```

Here SOCKET is the descriptor returned by socket() call and ADDRESS is a socket address (for TCP/IP) containing three elements −

- The address family (For TCP/IP, that's AF_INET, probably 2 on your system).
- The port number (for example 21).
- The internet address of the computer (for example 10.12.12.168).

As the bind() is used by a server, which does not need to know its own address so the argument list looks like this −

```
use Socket       # This defines PF_INET and SOCK_STREAM

$port = 12345;    # The unique port used by the sever to listen requests
$server_ip_address = "10.12.12.168";
bind( SOCKET, pack_sockaddr_in($port, inet_aton($server_ip_address)))
  or die "Can't bind to port $port! \n";
```

The **or die** clause is very important because if a server dies without outstanding connections, the port won't be immediately reusable unless you use the option SO_REUSEADDR using **setsockopt()** function. Here **pack_sockaddr_in()** function is being used to pack the Port and IP address into binary format.

The listen() call

If this is a server program, then it is required to issue a call to **listen()** on the specified port to listen, i.e., wait for the incoming requests. This call has the following syntax −

```
listen( SOCKET, QUEUESIZE );
```

The above call uses SOCKET descriptor returned by socket() call and QUEUESIZE is the maximum number of outstanding connection request allowed simultaneously.

The accept() call

If this is a server program then it is required to issue a call to the **access()** function to accept the incoming connections. This call has the following syntax −

```
accept( NEW_SOCKET, SOCKET );
```

The accept call receive SOCKET descriptor returned by socket() function and upon successful completion, a new socket descriptor NEW_SOCKET is returned for all future communication between the client and the server. If access() call fails, then it returns FLASE which is defined in Socket module which we have used initially.

Generally, accept() is used in an infinite loop. As soon as one connection arrives the server either creates a child process to deal with it or serves it himself and then goes back to listen for more connections.

```
while(1) {
  accept( NEW_SOCKET, SOCKT );
  .......
}
```

Now all the calls related to server are over and let us see a call which will be required by the client.

Client Side Socket Calls

The connect() call

If you are going to prepare client program, then first you will use **socket()** call to create a socket and then you would have to use **connect()** call to connect to the server. You already have seen socket() call syntax and it will remain similar to server socket() call, but here is the syntax for **connect()** call −

```
connect( SOCKET, ADDRESS );
```

Here SCOKET is the socket descriptor returned by socket() call issued by the client and ADDRESS is a socket address similar to *bind* call, except that it contains the IP address of the remote server.

```
$port = 21;    # For example, the ftp port
$server_ip_address = "10.12.12.168";
connect( SOCKET, pack_sockaddr_in($port, inet_aton($server_ip_address)))
   or die "Can't connect to port $port! \n";
```

If you connect to the server successfully, then you can start sending your commands to the server using SOCKET descriptor, otherwise your client will come out by giving an error message.

Client - Server Example

Following is a Perl code to implement a simple client-server program using Perl socket. Here server listens for incoming requests and once connection is established, it simply replies *Smile from the server*. The client reads that message and print on the screen. Let's see how it has been done, assuming we have our server and client on the same machine.

Script to Create a Server

```
#!/usr/bin/perl -w
# Filename : server.pl

use strict;
use Socket;

# use port 7890 as default
my $port = shift || 7890;
my $proto = getprotobyname('tcp');
my $server = "localhost";  # Host IP running the server

# create a socket, make it reusable
socket(SOCKET, PF_INET, SOCK_STREAM, $proto)
   or die "Can't open socket $!\n";
setsockopt(SOCKET, SOL_SOCKET, SO_REUSEADDR, 1)
   or die "Can't set socket option to SO_REUSEADDR $!\n";

# bind to a port, then listen
bind( SOCKET, pack_sockaddr_in($port, inet_aton($server)))
   or die "Can't bind to port $port! \n";
```

```
listen(SOCKET, 5) or die "listen: $!";
print "SERVER started on port $port\n";

# accepting a connection
my $client_addr;
while ($client_addr = accept(NEW_SOCKET, SOCKET)) {
   # send them a message, close connection
   my $name = gethostbyaddr($client_addr, AF_INET );
   print NEW_SOCKET "Smile from the server";
   print "Connection recieved from $name\n";
   close NEW_SOCKET;
}
```

To run the server in background mode issue the following command on Unix prompt −

```
$perl sever.pl&
```

Script to Create a Client

```
!/usr/bin/perl -w
# Filename : client.pl

use strict;
use Socket;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 7890;
my $server = "localhost";  # Host IP running the server

# create the socket, connect to the port
socket(SOCKET,PF_INET,SOCK_STREAM,(getprotobyname('tcp'))[2])
   or die "Can't create a socket $!\n";
connect( SOCKET, pack_sockaddr_in($port, inet_aton($server)))
   or die "Can't connect to port $port! \n";

my $line;
while ($line = <SOCKET>) {
   print "$line\n";
}
close SOCKET or die "close: $!";
```

Now let's start our client at the command prompt, which will connect to the server and read message sent by the server and displays the same on the screen as follows −

```
$perl client.pl
Smile from the server
```

NOTE − If you are giving the actual IP address in dot notation, then it is recommended to provide IP address in the same format in both client as well as server to avoid any confusion.