Exception Handling in Perl

Matt Sergeant

Exception Handling

- This talk is about exception handling
- But first we need to know why we have exception handling
- in the beginning, we had return codes
- 1 for success, 0 for failure
- ... or sometimes 0 for success and > 0 for failure
- ... and more often than not, a meaningful value for success, and 0 for failure

Error Codes

- Errors are a code as defined in errno.h
- Except when they're not
- You can get a meaningful description of the error with strerr() in C
- Sometimes you have to get the error code via a function call
- $! is even more fun
- $! is a string
- Except when you treat it as a number

System Calls (system, backticks, piped-open)

- System calls return 0 for success and > 0 for failure
- But their errors go into $!
- Sorry, into $?
- Actually different errors go into **each**
- And the return value from system() - that's the exit code from the child process.
- Nope. That's the return code multiplied by 256.
- There *are* some reasons for this, I assure you
- So to get the actual return code we divide by 256?
- Well the docs say to $? >> 8

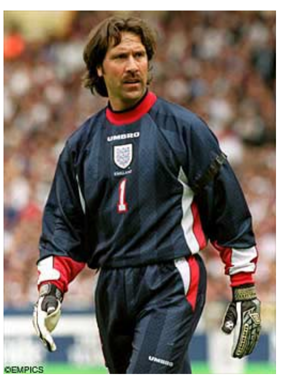
ARGHHH!!!

Insanity

- This of course is completely insane

Passing Return Codes

- Passing return codes up the stack is like a bucket brigage
- With every pass back up the call stack, you lose a bit of information
- It's also a lot of hard work!
- and it's too easy to get a butterfingers on your team



Outputting Errors

- With error codes we have two choices:
- Handle locally - outputting the error message
- Handle regionally or globally
- Local error handling is OK, but doesn't scale
- Regional error handling is more flexible...
- But the information loss means we punt:

```
print LOG "An error happened: $!"
```

- Where? What were the parameters when this happened? What was the call stack?

There is a better way

- Functions and methods should return what they are supposed to return **only**
- No more overloading of the return value
- We throw an *exception* when things go wrong
- This separates return values from error handling
  - SoC - Separation of Concerns

Exceptions

- Exceptions are really simple. We've all seen them:

```
open(FILE, $file) || die "Could not open $file: $!";
```

- But this is little use if our program can't exit
- So we can trap the exception:

```
eval {
  open(FILE, $file) || die "Could not open $file: $!";
  while(<FILE>) {
    process_line($_);
  }
  close(FILE) || die "Could not close $file: $!";
};
```

Exceptions - error message

- eval{} wraps the exception
- Error text goes in $@
- The error can occur at arbitrarily deep nesting levels
- die() is fast - implemented using setjmp/longjmp in C
- Test $@ to see if an exception occured:

```
eval {
  ...
};
if ($@) {
  handle_error($@);
}
```

Problems with $@ strings

- $@ errors are strings
- Matching what kind of error occured is error prone:

```
eval {
  open(FILE, $file) || die "Could not open $file: $!";
  process_file(\*FILE);
  close(FILE) || die "Could not close $file: $!";
};
if ($@) {
  if ($@ =~ m/Could not open/) {
    print "File open failure: $@";
  }
}
```

- This is great, until:

```
sub process_file {
  my $fh = shift;
  ...
  send_message();
}

sub send_message {
  open(MAIL, "| /usr/sbin/sendmail -t") || die "Could not open pipe to mail: $!";
  ...
}
```

$@ and context

- $@ doesn't tell us where the error occurred
- We can get around this with a custom function:

```
sub throw {
    my $mess = join('', @_);
    $mess =~ s/\n?$/\n/;
    my $i = 1;
    local $" = "', '";
    package DB;
    while (my @parts = caller($i++)) {
        my $q; $q = "'" if @DB::args;
        $mess .= " -> $parts[3](@DB::args)" .
          " at $parts[1] line $parts[2]\n";
    }
    die $mess;
}
```

throw output

```
sub main {
    my $args = shift;
    print "Hello\n";
    throw "Bad error\n";
}

main(\@ARGS);
```

```
Hello
Bad error
 -> main::main('ARRAY(0x8057b14)') at except1.pl line 19
```

- This, on its own, is incredibly useful
- See the Carp module for something very similar

Exception Objects

- Exceptions as strings have flaws:
  - They are not internationalised
  - They are not structured
- Perl 5.005 introduced Exception Objects

Exception Objects

```
eval {
  open(FILE, $file) ||
    die MyException::File->new();
};
if ($@) {
  # $@ contains the MyException::File object
}
```

- In MyException::File::new() we can do caller() to get context
- Now we can really test exception types again:

```
if ($@) {
  if ($@->isa('MyException::File')) {
    # handle File exception
  }
  else {
    # handle others
  }
}
```

- Exception objects implement stringification, for when they get printed or propogate to STDERR

$SIG{__DIE__} considered harmful

- $SIG{__DIE__} lets you set a handler for all die calls

```
$SIG{__DIE__} = sub { ...};
$SIG{__DIE__} = \&handle_die;
$SIG{__DIE__} = "handle_die";
```

- The handler is called when your program calls die()
- When your handler returns, the die() continues
- The only problem is it's a global - action at a distance
- Other perl modules erroneously set $SIG{__DIE__} to trap exceptions
  - e.g. older versions of CGI::Carp with qw(fatalsToBrowser)

Evil $SIG{__DIE__} example

```
$SIG{__DIE__} = sub {
  my $err = shift;
  warn("Caught error: ", $err);
  exit(55);
};

sub main {
  print "x: ";
  chomp(my $x = <STDIN>);
  print "y: ";
  chomp(my $y = <STDIN>);
  my $res = Foo::safe_divide($x, $y);
  print "Result: $res\n";
}

main();

package Foo;

sub safe_divide {
  my ($x, $y) = @_;
  my $result;
  eval {
    $result = $x / $y;
  };
  if ($@) {
    return undef;
  }
  return $result;
}
```

What is $SIG{__DIE__} good for?

- Let's assume we're throwing exception objects
- Your exception objects provide a nice stack trace using caller()
- But external modules that just die($string) don't
- Here's where $SIG{__DIE__} comes in

Useful $SIG{__DIE__} example

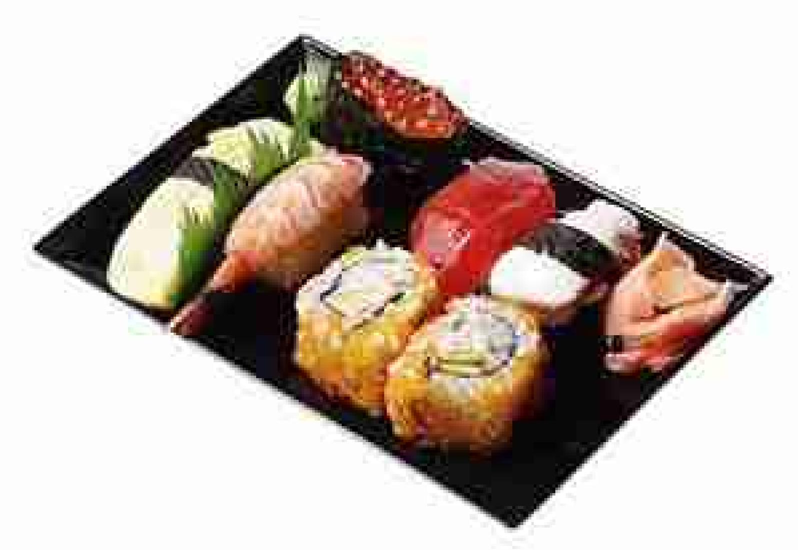
```
local $SIG{__DIE__} = sub {
  my $err = shift;
  if ($err->isa('MyException')) {
    die $err; # re-throw
  }
  else {
    # Otherwise construct a MyException with $err as a string
    die MyException::Default->new($err);
  }
};


sub main {
  eval {
    die "Fooey!";
  };
  if ($@) {
    if ($@->isa('MyException')) {
      warn("$0 died with:\n", $@->to_string);
    }
  }
}

main();
```

Exception Modules

- Error.pm
- Exception::Class
- Fatal.pm
- Not CGI::Carp



Error.pm

- The first CPAN implementor of exception objects
- Also implements try/catch/throw syntax:

```
try {
  open(FILE, "foo.txt") || throw Error::IO(
    -text => "File open failed: $!",
    -file => "foo.txt",
    );
  ...
}
catch Error::IO with {
  my $E = shift;
  print STDERR "File ", $E->{'-file'}, " had a problem\n";
  print STDERR $E;
}
finally {
  print "Program finished\n";
}
```

Error.pm stack traces

- Error.pm produces a stack trace if $Error::Debug is true

```
File foo.txt had a problem

File open failed: No such file or directory at except5.pl line 6
        main::main('HASH(0x804b584)') called at except5.pl line 22
        main::foo('a', 3) called at except5.pl line 25

Program finished
```

But...

- Error.pm works with a little bit of prototype magic:
- sub try (&;$) { ... }
- The & forces the first parameter to be a CODE ref
- This includes forcing blocks to become anonymous subs
- And if you reference variables outside the try block's scope, you get a closure

Closures...

- Closures are subroutines that hold onto a copy of variables declared outside their scope

```
{
   my $x = 1;
   sub foo { print $x++, "\n" };
}

# You would expect $x to be out of scope here (and maybe even foo())


for (1..10) {
   foo();
}
```

- Closures are one of the biggest sources of confusion in the perl world

Closures leaking

```
use Error qw(:try);

for (1..10) {
  my $x = Foo->new();
  try {
    try { print "$_: $x\n" };
  };
}

END { warn("interpreter exit\n") }

package Foo;
sub new { return bless {}, shift }
sub DESTROY { warn("Foo::DESTROY\n") }
```

- For a long-running daemon, this is a disaster

Error.pm Conclusions

- Because of the closure issue, I cannot recommend the try/catch syntax
- However the core classes are still good
- But there's a better set of error classes...

Exception::Class

- A base class and class-maker for derived exception classes

```
package Foo::Bar::Exceptions;

use Exception::Class (
        Foo::Bar::Exception::Senses => {
            description => 'sense-related exception',
        },
        Foo::Bar::Exception::Smell => {
            isa => 'Foo::Bar::Exception::Senses',
            fields => 'odor',
            description => 'stinky!',
        },
        Foo::Bar::Exception::Taste => {
            isa => 'Foo::Bar::Exception::Senses',
            fields => [ 'taste', 'bitterness' ],
            description => 'like, gag me with a spoon!',
        },
);
```

Exception::Class usage

```
eval {
  open(FILE, $file) || throw Foo::Bar::Exception::IO(
                                error => "File open failed: $!",
                                file => $file,
                               );
  ...
};
if ($@) {
  if ($@->isa('Foo::Bar::Exception::IO')) {
    print STDERR "IO Exception [", $@->file, "]: $@";
  }
}
```

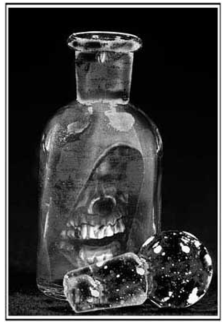
- Provides full structured stack trace with $@->trace and $@->trace->as_string
- Also provides process/user/group information with $@->uid, $@->gid, $@->pid, $@->time

Fatal.pm

- Fatal.pm overloads previously defined functions that return true/false
- This allows you to be lazy

```
use Fatal qw(open close chdir unlink socket);
use Socket;

eval { open(FILE, "/file/does/not/exist") }; warn($@) if $@;
eval { close(FILE) }; warn($@) if $@;
eval { chdir("wibblewibble") }; warn($@) if $@;
eval { unlink("/etc/passwd") }; warn($@) if $@;
eval { socket(SOCK, PF_INET, SOCK_STREAM, getprotobyname('tcp')) }; warn($
```



Fatal.pm Annoyances

- By default Fatal.pm catches every use of your functions

```
use Fatal qw(open);
if (open(FILE, $file)) {
   ...
}
else {
   # never get here
}
```

- Perl 5.6's Fatal.pm allows you to fix this:

```
use Fatal qw(:void open close);
```

Exceptions - Best Practices

- Use Exception::Class to define your exception classes
- Use Fatal.pm for core perl functions like open/close
- Define a $SIG{__DIE__} to turn string exceptions into objects
- Trap exceptions using eval{ ... }; if($@) { ... }
- ***Let exceptions propogate!***
- If you do need to handle an exception, make sure you only handle the types you expect:

```
if ($@) {
  if ($@->isa('MyException::Net')) {
    redo if $connect_attempts < 10;
  }
  $@->rethrow;
}
```

Exceptions - Best Practices (cont.)

- If you need to *log* all exceptions, wrap your main() in an exception handler
- If you can use Perl 5.6+, use Fatal qw(:void)
- Exception::Class allows descriptions for your exceptions. Use it.
- ***Don't return error codes. Please ;-)***