NAME

Guard – safe cleanup blocks

SYNOPSIS

```
use Guard;

# temporarily chdir to "/etc" directory, but make sure
# to go back to "/" no matter how myfun exits:
sub myfun {
   scope_guard { chdir "/" };
   chdir "/etc";

   code_that_might_die_or_does_other_fun_stuff;
}

# create an object that, when the last reference to it is gone,
# invokes the given codeblock:
my $guard = guard { print "destroyed!\n" };
undef $guard; # probably destroyed here
```

DESCRIPTION

This module implements so-called "guards". A guard is something (usually an object) that "guards" a resource, ensuring that it is cleaned up when expected.

Specifically, this module supports two different types of guards: guard objects, which execute a given code block when destroyed, and scoped guards, which are tied to the scope exit.

FUNCTIONS

This module currently exports the `scope_guard` and `guard` functions by default.

scope_guard BLOCK
scope_guard ($coderef)

Registers a block that is executed when the current scope (block, function, method, eval etc.) is exited.

See the EXCEPTIONS section for an explanation of how exceptions (i.e. `die`) are handled inside guard blocks.

The description below sounds a bit complicated, but that's just because `scope_guard` tries to get even corner cases "right": the goal is to provide you with a rock solid clean up tool.

The behaviour is similar to this code fragment:

```
eval ... code following scope_guard ...
{
   local $@;
   eval BLOCK;
   eval { $Guard::DIED->() } if $@;
}
die if $@;
```

Except it is much faster, and the whole thing gets executed even when the BLOCK calls `exit`, `goto`, `last` or escapes via other means.

If multiple BLOCKs are registered to the same scope, they will be executed in reverse order. Other scope-related things such as `local` are managed via the same mechanism, so variables `localised` *after* calling `scope_guard` will be restored when the guard runs.

Example: temporarily change the timezone for the current process, ensuring it will be reset when the `if` scope is exited:

```
use Guard;
use POSIX ();

if ($need_to_switch_tz) {
   # make sure we call tzset after $ENV{TZ} has been restored
   scope_guard { POSIX::tzset };

   # localise after the scope_guard, so it gets undone in time
   local $ENV{TZ} = "Europe/London";
   POSIX::tzset;

   # do something with the new timezone
}
```

my `$guard` = guard BLOCK
my `$guard` = guard ($coderef)

Behaves the same as `scope_guard`, except that instead of executing the block on scope exit, it returns an object whose lifetime determines when the BLOCK gets executed: when the last reference to the object gets destroyed, the BLOCK gets executed as with `scope_guard`.

See the EXCEPTIONS section for an explanation of how exceptions (i.e. `die`) are handled inside guard blocks.

Example: acquire a Coro::Semaphore for a second by registering a timer. The timer callback references the guard used to unlock it again. (Please ignore the fact that `Coro::Semaphore` has a `guard` method that does this already):

```
use Guard;
use Coro::AnyEvent;
use Coro::Semaphore;

my $sem = new Coro::Semaphore;

sub lock_for_a_second {
   $sem->down;
   my $guard = guard { $sem->up };

   Coro::AnyEvent::sleep 1;

   # $sem->up gets executed when returning
}
```

The advantage of doing this with a guard instead of simply calling `$sem->down` in the callback is that you can opt not to create the timer, or your code can throw an exception before it can create the timer (or the thread gets canceled), or you can create multiple timers or other event watchers and only when the last one gets executed will the lock be unlocked. Using the `guard`, you do not have to worry about catching all the places where you have to unlock the semaphore.

`$guard`->cancel

Calling this function will "disable" the guard object returned by the `guard` function, i.e. it will free the BLOCK originally passed to `guard` and will arrange for the BLOCK not to be executed.

This can be useful when you use `guard` to create a cleanup handler to be called under fatal conditions and later decide it is no longer needed.

EXCEPTIONS

Guard blocks should not normally throw exceptions (that is, `die`). After all, they are usually used to clean up after such exceptions. However, if something truly exceptional is happening, a guard block should of course be allowed to die. Also, programming errors are a large source of exceptions, and the programmer certainly wants to know about those.

Since in most cases, the block executing when the guard gets executed does not know or does not care about the guard blocks, it makes little sense to let containing code handle the exception.

Therefore, whenever a guard block throws an exception, it will be caught by Guard, followed by calling the code reference stored in `$Guard::DIED` (with `$@` set to the actual exception), which is similar to how most event loops handle this case.

The default for `$Guard::DIED` is to call `warn "$@"`, i.e. the error is printed as a warning and the program continues.

The `$@` variable will be restored to its value before the guard call in all cases, so guards will not disturb `$@` in any way.

The code reference stored in `$Guard::DIED` should not die (behaviour is not guaranteed, but right now, the exception will simply be ignored).

AUTHOR

```
 Marc Lehmann <schmorp@schmorp.de>
 http://home.schmorp.de/
```

THANKS

Thanks to Marco Maisenhelder, who reminded me of the `$Guard::DIED` solution to the problem of exceptions.

SEE ALSO

Scope::Guard and Sub::ScopeFinalizer, which actually implement dynamically scoped guards only, not the lexically scoped guards that their documentation promises, and have a lot higher CPU, memory and typing overhead.

Hook::Scope, which has apparently never been finished and can corrupt memory when used.

Scope::Guard seems to have a big SEE ALSO section for even more modules like it.