NAME

"Struct::Dumb" − make simple lightweight record−like structures

SYNOPSIS

```
use Struct::Dumb;

struct Point => [qw( x y )];

my $point = Point(10, 20);

printf "Point is at (%d, %d)\n", $point->x, $point->y;

$point->y = 30;
printf "Point is now at (%d, %d)\n", $point->x, $point->y;
struct Point3D => [qw( x y z )], named_constructor => 1;

my $point3d = Point3D( z => 12, x => 100, y => 50 );

printf "Point3d's height is %d\n", $point3d->z;
struct Point3D => [qw( x y z )], predicate => "is_Point3D";

my $point3d = Point3D( 1, 2, 3 );

printf "This is a Point3D\n" if is_Point3D( $point3d );
use Struct::Dumb qw( -named_constructors )

struct Point3D => [qw( x y z ];

my $point3d = Point3D( x => 100, z => 12, y => 50 );
```

DESCRIPTION

`Struct::Dumb` creates record-like structure types, similar to the `struct` keyword in C, C++ or C#, or `Record` in Pascal. An invocation of this module will create a construction function which returns new object references with the given field values. These references all respond to lvalue methods that access or modify the values stored.

It's specifically and intentionally not meant to be an object class. You cannot subclass it. You cannot provide additional methods. You cannot apply roles or mixins or metaclasses or traits or antlers or whatever else is in fashion this week.

On the other hand, it is tiny, creates cheap lightweight array-backed structures, uses nothing outside of core. It's intended simply to be a slightly nicer way to store data structures, where otherwise you might be tempted to abuse a hash, complete with the risk of typoing key names. The constructor will `croak` if passed the wrong number of arguments, as will attempts to refer to fields that don't exist. Accessor-mutators will `croak` if invoked with arguments. (This helps detect likely bugs such as accidentally passing in the new value as an argument, or attempting to invoke a stored `CODE` reference by passing argument values directly to the accessor.)

```
$ perl -E 'use Struct::Dumb; struct Point => [qw( x y )]; Point(30)'
usage: main::Point($x, $y) at -e line 1

$ perl -E 'use Struct::Dumb; struct Point => [qw( x y )]; Point(10,20)->z'
main::Point does not have a 'z' field at -e line 1

$ perl -E 'use Struct::Dumb; struct Point => [qw( x y )]; Point(1,2)->x(3)'
main::Point->x invoked with arguments at -e line 1.
```

Objects in this class are (currently) backed by an ARRAY reference store, though this is an internal implementation detail and should not be relied on by using code. Attempting to dereference the object as an ARRAY will throw an exception.

CONSTRUCTOR FORMS

The `struct` and `readonly_struct` declarations create two different kinds of constructor function, depending on the setting of the `named_constructor` option. When false, the constructor takes positional values in the same order as the fields were declared. When true, the constructor takes a key/value pair list in no particular order, giving the value of each named field.

This option can be specified to the `struct` and `readonly_struct` functions. It defaults to false, but it can be set on a per-package basis to default true by supplying the `-named_constructors` option on the `use` statement.

FUNCTIONS

struct

```
struct $name => [ @fieldnames ],
   named_constructor => (1|0),
   predicate         => "is_$name";
```

Creates a new structure type. This exports a new function of the type's name into the caller's namespace. Invoking this function returns a new instance of a type that implements those field names, as accessors and mutators for the fields.

Takes the following options:

named_constructor => BOOL
    Determines whether the structure will take positional or named arguments.

predicate => STR
    If defined, gives the name of a second function to export to the caller's namespace. This function will be a type test predicate; that is, a function that takes a single argmuent, and returns true if-and-only-if that argument is an instance of this structure type.

readonly_struct

```
readonly_struct $name => [ @fieldnames ],
   ...
```

Similar to "struct", but instances of this type are immutable once constructed. The field accessor methods will not be marked with the `:lvalue` attribute.

Takes the same options as "struct".

NOTES

Allowing ARRAY dereference

The way that forbidding access to instances as if they were ARRAY references is currently implemented uses an internal method on the generated structure class called `_forbid_arrayification`. If special circumstances require that this exception mechanism be bypassed, the method can be overloaded with an empty `sub {}` body, allowing the struct instances in that class to be accessed like normal ARRAY references. For good practice this should be limited by a `local` override.

For example, Devel::Cycle needs to access the instances as plain ARRAY references so it can walk the data structure looking for reference cycles.

```
use Devel::Cycle;

{
   no warnings 'redefine';
   local *Point::_forbid_arrayification = sub {};

   memory_cycle_ok( $point );
}
```

TODO

- Consider adding an `coerce_hash` option, giving name of another function to convert structs to key/value pairs, or a HASH ref.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>