NAME

fields - compile-time class fields

SYNOPSIS

```
{
package Foo;
use fields qw(foo bar _Foo_private);
sub new {
my Foo $self = shift;
unless (ref $self) {
$self = fields::new($self);
$self->{_Foo_private} = "this is Foo's secret";
}
$self->{foo} = 10;
$self->{bar} = 20;
return $self;
}
}

my $var = Foo->new;
$var->{foo} = 42;

# this will generate a run-time error
$var->{zap} = 42;

# this will generate a compile-time error
my Foo $foo = Foo->new;
$foo->{zap} = 24;

# subclassing
{
package Bar;
use base 'Foo';
use fields qw(baz _Bar_private); # not shared with Foo
sub new {
my $class = shift;
my $self = fields::new($class);
$self->SUPER::new(); # init base fields
$self->{baz} = 10; # init own fields
$self->{_Bar_private} = "this is Bar's secret";
return $self;
}
}
```

DESCRIPTION

The `fields` pragma enables compile-time and run-time verified class fields.

NOTE: The current implementation keeps the declared fields in the `%FIELDS` hash of the calling package, but this may change in future versions. Do**not** update the `%FIELDS` hash directly , because it must be created at compile-time for it to be fully useful, as is done by this pragma.

If a typed lexical variable (`my Class $var`) holding a reference is used to access a hash element and a package with the same name as the type has declared class fields using this pragma, then the hash key is verified at compile time. If the variables are not typed, access is only checked at run time.

The related `base` pragma will combine fields from base classes and any fields declared using the

`fields` pragma. This enables field inheritance to work properly. Inherited fields can be overridden but will generate a warning if warnings are enabled.

Only valid for Perl 5.8.x and earlier: Field names that start with an underscore c haracter are made private to the class and are not visible to subclasses.

Also, **in Perl 5.8.x and earlier**, this pragma uses pseudo-hashes, the effect being that you can have objects with named fields which are as compact and as fast arrays to access, as long as the objects are accessed through properly typed variables.

The following functions are supported:

new

fields::new() creates and blesses a hash comprised of the fields declared using the `fields` pragma into the specified class. It is the recommended way to construct a fields-based object.

This makes it possible to write a constructor like this:

```
package Critter::Sounds;
use fields qw(cat dog bird);

sub new {
my $self = shift;
$self = fields::new($self) unless ref $self;
$self->{cat} = 'meow'; # scalar element
@$self{'dog','bird'} = ('bark','tweet'); # slice
return $self;
}
```

phash

This function only works in Perl 5.8.x and earlier. Pseudo-hashes w ere removed from Perl as of 5.10. Consider using restricted hashes or *fields::new()* instead (which itself uses restricted hashes under 5.10+). See Hash::Util. Using *fields::phash()* under 5.10 or higher will cause an error.

fields::phash() can be used to create and initialize a plain (unblessed) pseudo-hash. This function should always be used instead of creating pseudo-hashes directly.

If the first argument is a reference to an array, the pseudo-hash will be created with keys from that array. If a second argument is supplied, it must also be a reference to an array whose elements will be used as the values. If the second array contains less elements than the first, the trailing elements of the pseudo-hash will not be initialized. This makes it particularly useful for creating a pseudo-hash from subroutine arguments:

```
sub dogtag {
my $tag = fields::phash([qw(name rank ser_num)], [@_]);
}
```

fields::phash() also accepts a list of key-value pairs that will be used to construct the pseudo hash. Examples:

```
my $tag = fields::phash(name => "Joe",
rank => "captain",
ser_num => 42);

my $pseudohash = fields::phash(%args);
```

SEE ALSO

base, Hash::Util