NAME

constant - Perl pragma to declare constants

SYNOPSIS

```
use constant PI    => 4 * atan2(1, 1);
use constant DEBUG => 0;

print "Pi equals ", PI, "...\n" if DEBUG;

use constant {
    SEC   => 0,
    MIN   => 1,
    HOUR  => 2,
    MDAY  => 3,
    MON   => 4,
    YEAR  => 5,
    WDAY  => 6,
    YDAY  => 7,
    ISDST => 8,
};

use constant WEEKDAYS => qw(
    Sunday Monday Tuesday Wednesday Thursday Friday Saturday
);

print "Today is ", (WEEKDAYS)[ (localtime)[WDAY] ], ".\n";
```

DESCRIPTION

This will declare a symbol to be a constant with the given value.

When you declare a constant such as `PI` using the method shown above, each machine your script runs upon can have as many digits of accuracy as it can use. Also, your program will be easier to read, more likely to be maintained (and maintained correctly), and far less likely to send a space probe to the wrong planet because nobody noticed the one equation in which you wrote `3.14195`.

When a constant is used in an expression, perl replaces it with its value at compile time, and may then optimize the expression further. In particular, any code in an `if (CONSTANT)` block will be optimized away if the constant is false.

NOTES

As with all `use` directives, defining a constant happens at compile time. Thus, it's probably not correct to put a constant declaration inside of a conditional statement (like `if ($foo) { use constant ... }`).

Constants defined using this module cannot be interpolated into strings like variables. However, concatenation works just fine:

```
print "Pi equals PI...\n";        # WRONG: does not expand "PI"
print "Pi equals ".PI."...\n";    # right
```

Even though a reference may be declared as a constant, the reference may point to data which may be changed, as this code shows.

```
use constant ARRAY => [ 1,2,3,4 ];
print ARRAY->[1];
```

```
ARRAY->[1] = " be changed";
print ARRAY->[1];
```

Dereferencing constant references incorrectly (such as using an array subscript on a constant hash reference, or vice versa) will be trapped at compile time.

Constants belong to the package they are defined in. To refer to a constant defined in another package, specify the full package name, as in `Some::Package::CONSTANT`. Constants may be exported by modules, and may also be called as either class or instance methods, that is, as `Some::Package->CONSTANT` or as `$obj->CONSTANT` where `$obj` is an instance of `Some::Package`. Subclasses may define their own constants to override those in their base class.

The use of all caps for constant names is merely a convention, although it is recommended in order to make constants stand out and to help avoid collisions with other barewords, keywords, and subroutine names. Constant names must begin with a letter or underscore. Names beginning with a double underscore are reserved. Some poor choices for names will generate warnings, if warnings are enabled at compile time.

List constants

Constants may be lists of more (or less) than one value. A constant with no values evaluates to `undef` in scalar context. Note that constants with more than one value do *not* return their last value in scalar context as one might expect. They currently return the number of values, but **this may change in the future**. Do not use constants with multiple values in scalar context.

NOTE: This implies that the expression defining the value of a constant is evaluated in list context. This may produce surprises:

```
use constant TIMESTAMP => localtime;                # WRONG!
use constant TIMESTAMP => scalar localtime;         # right
```

The first line above defines `TIMESTAMP` as a 9-element list, as returned by localtime() in list context. To set it to the string returned by localtime() in scalar context, an explicit `scalar` keyword is required.

List constants are lists, not arrays. To index or slice them, they must be placed in parentheses.

```
my @workdays = WEEKDAYS[1 .. 5];            # WRONG!
my @workdays = (WEEKDAYS)[1 .. 5];          # right
```

Defining multiple constants at once

Instead of writing multiple `use constant` statements, you may define multiple constants in a single statement by giving, instead of the constant name, a reference to a hash where the keys are the names of the constants to be defined. Obviously, all constants defined using this method must have a single value.

```
use constant {
    FOO => "A single value",
    BAR => "This", "won't", "work!",        # Error!
};
```

This is a fundamental limitation of the way hashes are constructed in Perl. The error messages produced when this happens will often be quite cryptic -- in the worst case there may be none at all, and you'll only later find that something is broken.

When defining multiple constants, you cannot use the values of other constants defined in the same declaration. This is because the calling package doesn't know about any constant within that group until *after* the `use` statement is finished.

```
    use constant {
        BITMASK => 0xAFBAEBA8,
        NEGMASK => ~BITMASK,                    # Error!
    };
```

Magic constants

Magical values and references can be made into constants at compile time, allowing for way cool stuff like this. (These error numbers aren't totally portable, alas.)

```
    use constant E2BIG => ($! = 7);
    print   E2BIG, "\n";        # something like "Arg list too long"
    print 0+E2BIG, "\n";        # "7"
```

You can't produce a tied constant by giving a tied scalar as the value. References to tied variables, however, can be used as constants without any problems.

TECHNICAL NOTES

In the current implementation, scalar constants are actually inlinable subroutines. As of version 5.004 of Perl, the appropriate scalar constant is inserted directly in place of some subroutine calls, thereby saving the overhead of a subroutine call. See *"Constant Functions" in perlsub* for details about how and when this happens.

In the rare case in which you need to discover at run time whether a particular constant has been declared via this module, you may use this function to examine the hash `%constant::declared`. If the given constant name does not include a package name, the current package is used.

```
    sub declared ($) {
        use constant 1.01;              # don't omit this!
        my $name = shift;
        $name =~ s/^::/main::/;
        my $pkg = caller;
        my $full_name = $name =~ /::/ ? $name : "${pkg}::$name";
        $constant::declared{$full_name};
    }
```

BUGS

In the current version of Perl, list constants are not inlined and some symbols may be redefined without generating a warning.

It is not possible to have a subroutine or a keyword with the same name as a constant in the same package. This is probably a Good Thing.

A constant with a name in the list `STDIN STDOUT STDERR ARGV ARGVOUT ENV INC SIG` is not allowed anywhere but in package `main::`, for technical reasons.

Unlike constants in some languages, these cannot be overridden on the command line or via environment variables.

You can get into trouble if you use constants in a context which automatically quotes barewords (as is true for any subroutine call). For example, you can't say `$hash{CONSTANT}` because `CONSTANT` will be interpreted as a string. Use `$hash{CONSTANT()}` or `$hash{+CONSTANT}` to prevent the bareword quoting mechanism from kicking in. Similarly, since the `=>` operator quotes a bareword immediately to its left, you have to say `CONSTANT() => 'value'` (or simply use a comma in place of the big arrow) instead of `CONSTANT => 'value'`.

AUTHOR

Tom Phoenix, <*rootbeer@redcat.com*>, with help from many other folks.

Multiple constant declarations at once added by Casey West, <*casey@geeknest.com*>.

Documentation mostly rewritten by Ilmari Karonen, <*perl@itz.pp.sci.fi*>.

COPYRIGHT