NAME

Time::Piece - Object Oriented time objects

SYNOPSIS

```
use Time::Piece;

my $t = localtime;
print "Time is $t\n";
print "Year is ", $t->year, "\n";
```

DESCRIPTION

This module replaces the standard `localtime` and `gmtime` functions with implementations that return objects. It does so in a backwards compatible manner, so that using localtime/gmtime in the way documented in perlfunc will still return what you expect.

The module actually implements most of an interface described by Larry Wall on the perl5-porters mailing list here: http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2000-01/msg00241.html

USAGE

After importing this module, when you use localtime or gmtime in a scalar context, rather than getting an ordinary scalar string representing the date and time, you get a Time::Piece object, whose stringification happens to produce the same effect as the localtime and gmtime functions. There is also a new() constructor provided, which is the same as localtime(), except when passed a Time::Piece object, in which case it's a copy constructor. The following methods are available on the object:

```
$t->sec                 # also available as $t->second
$t->min                 # also available as $t->minute
$t->hour                # 24 hour
$t->mday                # also available as $t->day_of_month
$t->mon                 # 1 = January
$t->_mon                # 0 = January
$t->monname             # Feb
$t->month               # same as $t->monname
$t->fullmonth           # February
$t->year                # based at 0 (year 0 AD is, of course 1 BC)
$t->_year               # year minus 1900
$t->yy                  # 2 digit year
$t->wday                # 1 = Sunday
$t->_wday               # 0 = Sunday
$t->day_of_week         # 0 = Sunday
$t->wdayname            # Tue
$t->day                 # same as wdayname
$t->fullday             # Tuesday
$t->yday                # also available as $t->day_of_year, 0 = Jan 01
$t->isdst               # also available as $t->daylight_savings

$t->hms                 # 12:34:56
$t->hms(".")            # 12.34.56
$t->time                # same as $t->hms

$t->ymd                 # 2000-02-29
$t->date                # same as $t->ymd
$t->mdy                 # 02-29-2000
$t->mdy("/")            # 02/29/2000
```

```
    $t->dmy                 # 29-02-2000
    $t->dmy(".")            # 29.02.2000
    $t->datetime            # 2000-02-29T12:34:56 (ISO 8601)
    $t->cdate               # Tue Feb 29 12:34:56 2000
    "$t"                    # same as $t->cdate

    $t->epoch               # seconds since the epoch
    $t->tzoffset            # timezone offset in a Time::Seconds object

    $t->julian_day          # number of days since Julian period began
    $t->mjd                 # modified Julian date (JD-2400000.5 days)

    $t->week                # week number (ISO 8601)

    $t->is_leap_year        # true if it's a leap year
    $t->month_last_day      # 28-31

    $t->time_separator($s)  # set the default separator (default ":")
    $t->date_separator($s)  # set the default separator (default "-")
    $t->day_list(@days)     # set the default weekdays
    $t->mon_list(@days)     # set the default months

    $t->strftime(FORMAT)    # same as POSIX::strftime (without the overhead
                            # of the full POSIX extension)
    $t->strftime()          # "Tue, 29 Feb 2000 12:34:56 GMT"

    Time::Piece->strptime(STRING, FORMAT)
                            # see strptime man page. Creates a new
                            # Time::Piece object
```

Note that `localtime` and `gmtime` are not listed above. If called as methods on a Time::Piece object, they act as constructors, returning a new Time::Piece object for the current time. In other words: they're not useful as methods.

Local Locales

Both wdayname (day) and monname (month) allow passing in a list to use to index the name of the days against. This can be useful if you need to implement some form of localisation without actually installing or using locales.

```
  my @days = qw( Dimanche Lundi Merdi Mercredi Jeudi Vendredi Samedi );

  my $french_day = localtime->day(@days);
```

These settings can be overridden globally too:

```
  Time::Piece::day_list(@days);
```

Or for months:

```
  Time::Piece::mon_list(@months);
```

And locally for months:

```
print localtime->month(@months);
```

Date Calculations

It's possible to use simple addition and subtraction of objects:

```
use Time::Seconds;

my $seconds = $t1 - $t2;
$t1 += ONE_DAY; # add 1 day (constant from Time::Seconds)
```

The following are valid ($t1 and $t2 are Time::Piece objects):

```
$t1 - $t2; # returns Time::Seconds object
$t1 - 42; # returns Time::Piece object
$t1 + 533; # returns Time::Piece object
```

However adding a Time::Piece object to another Time::Piece object will cause a runtime error.

Note that the first of the above returns a Time::Seconds object, so while examining the object will print the number of seconds (because of the overloading), you can also get the number of minutes, hours, days, weeks and years in that delta, using the Time::Seconds API.

In addition to adding seconds, there are two APIs for adding months and years:

```
$t->add_months(6);
$t->add_years(5);
```

The months and years can be negative for subtractions. Note that there is some "strange" behaviour when adding and subtracting months at the ends of months. Generally when the resulting month is shorter than the starting month then the number of overlap days is added. For example subtracting a month from 2008-03-31 will not result in 2008-02-31 as this is an impossible date. Instead you will get 2008-03-02. This appears to be consistent with other date manipulation tools.

Date Comparisons

Date comparisons are also possible, using the full suite of "<", ">", "<=", ">=", "<=>", "==" and "!=".

Date Parsing

Time::Piece has a built-in strptime() function (from FreeBSD), allowing you incredibly flexible date parsing routines. For example:

```
my $t = Time::Piece->strptime("Sunday 3rd Nov, 1943",
                              "%A %drd %b, %Y");

print $t->strftime("%a, %d %b %Y");
```

Outputs:

```
Wed, 03 Nov 1943
```

(see, it's even smart enough to fix my obvious date bug)

For more information see "man strptime", which should be on all unix systems.

Alternatively look here: http://www.unix.com/man-page/FreeBSD/3/strftime/

YYYY-MM-DDThh:mm:ss

The ISO 8601 standard defines the date format to be YYYY-MM-DD, and the time format to be hh:mm:ss (24 hour clock), and if combined, they should be concatenated with date first and with a capital 'T' in front of the time.

Week Number

The *week number* may be an unknown concept to some readers. The ISO 8601 standard defines that weeks begin on a Monday and week 1 of the year is the week that includes both January 4th and the first Thursday of the year. In other words, if the first Monday of January is the 2nd, 3rd, or 4th, the preceding days of the January are part of the last week of the preceding year. Week numbers range from 1 to 53.

Global Overriding

Finally, it's possible to override localtime and gmtime everywhere, by including the ':override' tag in the import list:

```
use Time::Piece ':override';
```

CAVEATS

Setting $ENV{TZ} in Threads on Win32

Note that when using perl in the default build configuration on Win32 (specifically, when perl is built with PERL_IMPLICIT_SYS), each perl interpreter maintains its own copy of the environment and only the main interpreter will update the process environment seen by strftime.

Therefore, if you make changes to $ENV{TZ} from inside a thread other than the main thread then those changes will not be seen by strftime if you subsequently call that with the %Z formatting code. You must change $ENV{TZ} in the main thread to have the desired effect in this case (and you must also call _tzset() in the main thread to register the environment change).

Furthermore, remember that this caveat also applies to fork(), which is emulated by threads on Win32.

Use of epoch seconds

This module internally uses the epoch seconds system that is provided via the perl `time()` function and supported by `gmtime()` and `localtime()`.

If your perl does not support times larger than 2^31 seconds then this module is likely to fail at processing dates beyond the year 2038. There are moves afoot to fix that in perl. Alternatively use 64 bit perl. Or if none of those are options, use the *DateTime* module which has support for years well into the future and past.

AUTHOR

Matt Sergeant, matt@sergeant.org Jarkko Hietaniemi, jhi@iki.fi (while creating Time::Piece for core perl)

COPYRIGHT AND LICENSE

SEE ALSO

The excellent Calendar FAQ at http://www.tondering.dk/claus/calendar.html

BUGS

The test harness leaves much to be desired. Patches welcome.