NAME

Tie::File - Access the lines of a disk file via a Perl array

SYNOPSIS

```
# This file documents Tie::File version 0.97
use Tie::File;

tie @array, 'Tie::File', filename or die ...;

$array[13] = 'blah';     # line 13 of the file is now 'blah'
print $array[42];        # display line 42 of the file

$n_recs = @array;        # how many records are in the file?
$#array -= 2;            # chop two records off the end

for (@array) {
  s/PERL/Perl/g;         # Replace PERL with Perl everywhere in the file
}

# These are just like regular push, pop, unshift, shift, and splice
# Except that they modify the file in the way you would expect

push @array, new recs...;
my $r1 = pop @array;
unshift @array, new recs...;
my $r2 = shift @array;
@old_recs = splice @array, 3, 7, new recs...;

untie @array;            # all finished
```

DESCRIPTION

`Tie::File` represents a regular text file as a Perl array. Each element in the array corresponds to a record in the file. The first line of the file is element 0 of the array; the second line is element 1, and so on.

The file is *not* loaded into memory, so this will work even for gigantic files.

Changes to the array are reflected in the file immediately.

Lazy people and beginners may now stop reading the manual.

recsep

What is a 'record'? By default, the meaning is the same as for the `<...>` operator: It's a string terminated by `$/`, which is probably `"\n"`. (Minor exception: on DOS and Win32 systems, a 'record' is a string terminated by `"\r\n"`.) You may change the definition of "record" by supplying the `recsep` option in the `tie` call:

```
tie @array, 'Tie::File', $file, recsep => 'es';
```

This says that records are delimited by the string `es`. If the file contained the following data:

```
Curse these pesky flies!\n
```

then the `@array` would appear to have four elements:

```
"Curse th"
"e p"
"ky fli"
"!\n"
```

An undefined value is not permitted as a record separator. Perl's special "paragraph mode" semantics (à la `$/ = ""`) are not emulated.

Records read from the tied array do not have the record separator string on the end; this is to allow

```
$array[17] .= "extra";
```

to work as expected.

(See *autochomp*, below.) Records stored into the array will have the record separator string appended before they are written to the file, if they don't have one already. For example, if the record separator string is `"\n"`, then the following two lines do exactly the same thing:

```
$array[17] = "Cherry pie";
$array[17] = "Cherry pie\n";
```

The result is that the contents of line 17 of the file will be replaced with "Cherry pie"; a newline character will separate line 17 from line 18. This means that this code will do nothing:

```
chomp $array[17];
```

Because the `chomp`ed value will have the separator reattached when it is written back to the file. There is no way to create a file whose trailing record separator string is missing.

Inserting records that *contain* the record separator string is not supported by this module. It will probably produce a reasonable result, but what this result will be may change in a future version. Use 'splice' to insert records or to replace one record with several.

autochomp

Normally, array elements have the record separator removed, so that if the file contains the text

```
Gold
Frankincense
Myrrh
```

the tied array will appear to contain `("Gold", "Frankincense", "Myrrh")`. If you set `autochomp` to a false value, the record separator will not be removed. If the file above was tied with

```
tie @gifts, "Tie::File", $gifts, autochomp => 0;
```

then the array `@gifts` would appear to contain `("Gold\n", "Frankincense\n", "Myrrh\n")`, or (on Win32 systems) `("Gold\r\n", "Frankincense\r\n", "Myrrh\r\n")`.

mode

Normally, the specified file will be opened for read and write access, and will be created if it does not exist. (That is, the flags `O_RDWR | O_CREAT` are supplied in the `open` call.) If you want to change this, you may supply alternative flags in the `mode` option. See *Fcntl* for a listing of available flags. For example:

```
# open the file if it exists, but fail if it does not exist
use Fcntl 'O_RDWR';
tie @array, 'Tie::File', $file, mode => O_RDWR;
```

```
# create the file if it does not exist
use Fcntl 'O_RDWR', 'O_CREAT';
tie @array, 'Tie::File', $file, mode => O_RDWR | O_CREAT;

# open an existing file in read-only mode
use Fcntl 'O_RDONLY';
tie @array, 'Tie::File', $file, mode => O_RDONLY;
```

Opening the data file in write-only or append mode is not supported.

memory

This is an upper limit on the amount of memory that `Tie::File` will consume at any time while managing the file. This is used for two things: managing the *read cache* and managing the *deferred write buffer*.

Records read in from the file are cached, to avoid having to re-read them repeatedly. If you read the same record twice, the first time it will be stored in memory, and the second time it will be fetched from the *read cache*. The amount of data in the read cache will not exceed the value you specified for `memory`. If `Tie::File` wants to cache a new record, but the read cache is full, it will make room by expiring the least-recently visited records from the read cache.

The default memory limit is 2Mib. You can adjust the maximum read cache size by supplying the `memory` option. The argument is the desired cache size, in bytes.

```
# I have a lot of memory, so use a large cache to speed up access
tie @array, 'Tie::File', $file, memory => 20_000_000;
```

Setting the memory limit to 0 will inhibit caching; records will be fetched from disk every time you examine them.

The `memory` value is not an absolute or exact limit on the memory used. `Tie::File` objects contains some structures besides the read cache and the deferred write buffer, whose sizes are not charged against `memory`.

The cache itself consumes about 310 bytes per cached record, so if your file has many short records, you may want to decrease the cache memory limit, or else the cache overhead may exceed the size of the cached data.

dw_size

(This is an advanced feature. Skip this section on first reading.)

If you use deferred writing (See *Deferred Writing*, below) then data you write into the array will not be written directly to the file; instead, it will be saved in the *deferred write buffer* to be written out later. Data in the deferred write buffer is also charged against the memory limit you set with the `memory` option.

You may set the `dw_size` option to limit the amount of data that can be saved in the deferred write buffer. This limit may not exceed the total memory limit. For example, if you set `dw_size` to 1000 and `memory` to 2500, that means that no more than 1000 bytes of deferred writes will be saved up. The space available for the read cache will vary, but it will always be at least 1500 bytes (if the deferred write buffer is full) and it could grow as large as 2500 bytes (if the deferred write buffer is empty.)

If you don't specify a `dw_size`, it defaults to the entire memory limit.

Option Format

`-mode` is a synonym for `mode`. `-recsep` is a synonym for `recsep`. `-memory` is a synonym for `memory`. You get the idea.

Public Methods

The `tie` call returns an object, say `$o`. You may call

```
$rec = $o->FETCH($n);
$o->STORE($n, $rec);
```

to fetch or store the record at line `$n`, respectively; similarly the other tied array methods. (See *perltie* for details.) You may also call the following methods on this object:

flock

```
$o->flock(MODE)
```

will lock the tied file. `MODE` has the same meaning as the second argument to the Perl built-in `flock` function; for example `LOCK_SH` or `LOCK_EX | LOCK_NB`. (These constants are provided by the `use Fcntl ':flock'` declaration.)

`MODE` is optional; the default is `LOCK_EX`.

`Tie::File` maintains an internal table of the byte offset of each record it has seen in the file.

When you use `flock` to lock the file, `Tie::File` assumes that the read cache is no longer trustworthy, because another process might have modified the file since the last time it was read. Therefore, a successful call to `flock` discards the contents of the read cache and the internal record offset table.

`Tie::File` promises that the following sequence of operations will be safe:

```
my $o = tie @array, "Tie::File", $filename;
$o->flock;
```

In particular, `Tie::File` will *not* read or write the file during the `tie` call. (Exception: Using `mode => O_TRUNC` will, of course, erase the file during the `tie` call. If you want to do this safely, then open the file without `O_TRUNC`, lock the file, and use `@array = ()`.)

The best way to unlock a file is to discard the object and untie the array. It is probably unsafe to unlock the file without also untying it, because if you do, changes may remain unwritten inside the object. That is why there is no shortcut for unlocking. If you really want to unlock the file prematurely, you know what to do; if you don't know what to do, then don't do it.

All the usual warnings about file locking apply here. In particular, note that file locking in Perl is **advisory**, which means that holding a lock will not prevent anyone else from reading, writing, or erasing the file; it only prevents them from getting another lock at the same time. Locks are analogous to green traffic lights: If you have a green light, that does not prevent the idiot coming the other way from plowing into you sideways; it merely guarantees to you that the idiot does not also have a green light at the same time.

autochomp

```
my $old_value = $o->autochomp(0);    # disable autochomp option
my $old_value = $o->autochomp(1);    #  enable autochomp option

my $ac = $o->autochomp();   # recover current value
```

See *autochomp*, above.

defer, flush, discard, and autodefer

See *Deferred Writing*, below.

offset

```
$off = $o->offset($n);
```

This method returns the byte offset of the start of the $nth record in the file. If there is no such record, it returns an undefined value.

Tying to an already-opened filehandle

If `$fh` is a filehandle, such as is returned by `IO::File` or one of the other `IO` modules, you may use:

```
tie @array, 'Tie::File', $fh, ...;
```

Similarly if you opened that handle `FH` with regular `open` or `sysopen`, you may use:

```
tie @array, 'Tie::File', \*FH, ...;
```

Handles that were opened write-only won't work. Handles that were opened read-only will work as long as you don't try to modify the array. Handles must be attached to seekable sources of data---that means no pipes or sockets. If `Tie::File` can detect that you supplied a non-seekable handle, the `tie` call will throw an exception. (On Unix systems, it can detect this.)

Note that Tie::File will only close any filehandles that it opened internally. If you passed it a filehandle as above, you "own" the filehandle, and are responsible for closing it after you have untied the @array.

Deferred Writing

(This is an advanced feature. Skip this section on first reading.)

Normally, modifying a `Tie::File` array writes to the underlying file immediately. Every assignment like `$a[3] = ...` rewrites as much of the file as is necessary; typically, everything from line 3 through the end will need to be rewritten. This is the simplest and most transparent behavior. Performance even for large files is reasonably good.

However, under some circumstances, this behavior may be excessively slow. For example, suppose you have a million-record file, and you want to do:

```
for (@FILE) {
  $_ = "> $_";
}
```

The first time through the loop, you will rewrite the entire file, from line 0 through the end. The second time through the loop, you will rewrite the entire file from line 1 through the end. The third time through the loop, you will rewrite the entire file from line 2 to the end. And so on.

If the performance in such cases is unacceptable, you may defer the actual writing, and then have it done all at once. The following loop will perform much better for large files:

```
(tied @a)->defer;
for (@a) {
  $_ = "> $_";
}
(tied @a)->flush;
```

If `Tie::File`'s memory limit is large enough, all the writing will done in memory. Then, when you call `->flush`, the entire file will be rewritten in a single pass.

(Actually, the preceding discussion is something of a fib. You don't need to enable deferred writing to get good performance for this common case, because `Tie::File` will do it for you automatically

unless you specifically tell it not to. See *autodeferring*, below.)

Calling `->flush` returns the array to immediate-write mode. If you wish to discard the deferred writes, you may call `->discard` instead of `->flush`. Note that in some cases, some of the data will have been written already, and it will be too late for `->discard` to discard all the changes. Support for `->discard` may be withdrawn in a future version of `Tie::File`.

Deferred writes are cached in memory up to the limit specified by the `dw_size` option (see above). If the deferred-write buffer is full and you try to write still more deferred data, the buffer will be flushed. All buffered data will be written immediately, the buffer will be emptied, and the now-empty space will be used for future deferred writes.

If the deferred-write buffer isn't yet full, but the total size of the buffer and the read cache would exceed the `memory` limit, the oldest records will be expired from the read cache until the total size is under the limit.

`push`, `pop`, `shift`, `unshift`, and `splice` cannot be deferred. When you perform one of these operations, any deferred data is written to the file and the operation is performed immediately. This may change in a future version.

If you resize the array with deferred writing enabled, the file will be resized immediately, but deferred records will not be written. This has a surprising consequence: `@a = (...)` erases the file immediately, but the writing of the actual data is deferred. This might be a bug. If it is a bug, it will be fixed in a future version.

Autodeferring

`Tie::File` tries to guess when deferred writing might be helpful, and to turn it on and off automatically.

```
for (@a) {
  $_ = "> $_";
}
```

In this example, only the first two assignments will be done immediately; after this, all the changes to the file will be deferred up to the user-specified memory limit.

You should usually be able to ignore this and just use the module without thinking about deferring. However, special applications may require fine control over which writes are deferred, or may require that all writes be immediate. To disable the autodeferment feature, use

```
(tied @o)->autodefer(0);
```

or

```
        tie @array, 'Tie::File', $file, autodefer => 0;
```

Similarly, `->autodefer(1)` re-enables autodeferment, and `->autodefer()` recovers the current value of the autodefer setting.

CONCURRENT ACCESS TO FILES

Caching and deferred writing are inappropriate if you want the same file to be accessed simultaneously from more than one process. Other optimizations performed internally by this module are also incompatible with concurrent access. A future version of this module will support a `concurrent => 1` option that enables safe concurrent access.

Previous versions of this documentation suggested using `memory => 0` for safe concurrent access. This was mistaken. Tie::File will not support safe concurrent access before version 0.98.

CAVEATS

(That's Latin for 'warnings'.)

- Reasonable effort was made to make this module efficient. Nevertheless, changing the size of a record in the middle of a large file will always be fairly slow, because everything after the new record must be moved.

- The behavior of tied arrays is not precisely the same as for regular arrays. For example:

```
# This DOES print "How unusual!"
undef $a[10];  print "How unusual!\n" if defined $a[10];
```

  `undef`-ing a `Tie::File` array element just blanks out the corresponding record in the file. When you read it back again, you'll get the empty string, so the supposedly-`undef`'ed value will be defined. Similarly, if you have `autochomp` disabled, then

```
# This DOES print "How unusual!" if 'autochomp' is disabled
undef $a[10];
       print "How unusual!\n" if $a[10];
```

  Because when `autochomp` is disabled, `$a[10]` will read back as `"\n"` (or whatever the record separator string is.)

  There are other minor differences, particularly regarding `exists` and `delete`, but in general, the correspondence is extremely close.

- I have supposed that since this module is concerned with file I/O, almost all normal use of it will be heavily I/O bound. This means that the time to maintain complicated data structures inside the module will be dominated by the time to actually perform the I/O. When there was an opportunity to spend CPU time to avoid doing I/O, I usually tried to take it.

- You might be tempted to think that deferred writing is like transactions, with `flush` as `commit` and `discard` as `rollback`, but it isn't, so don't.

- There is a large memory overhead for each record offset and for each cache entry: about 310 bytes per cached data record, and about 21 bytes per offset table entry.

  The per-record overhead will limit the maximum number of records you can access per file. Note that *accessing* the length of the array via `$x = scalar @tied_file` accesses **all** records and stores their offsets. The same for `foreach (@tied_file)`, even if you exit the loop early.

SUBCLASSING

This version promises absolutely nothing about the internals, which may change without notice. A future version of the module will have a well-defined and stable subclassing API.

WHAT ABOUT DB_File?

People sometimes point out that *DB_File* will do something similar, and ask why `Tie::File` module is necessary.

There are a number of reasons that you might prefer `Tie::File`. A list is available at `http://perl.plover.com/TieFile/why-not-DB_File`.

AUTHOR

Mark Jason Dominus

To contact the author, send email to: `mjd-perl-tiefile+@plover.com`

To receive an announcement whenever a new version of this module is released, send a blank email message to `mjd-perl-tiefile-subscribe@plover.com`.

The most recent version of this module, including documentation and any news of importance, will be available at

```
http://perl.plover.com/TieFile/
```

LICENSE

`Tie::File` version 0.97 is copyright (C) 2003 Mark Jason Dominus.

This library is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

These terms are your choice of any of (1) the Perl Artistic Licence, or (2) version 2 of the GNU General Public License as published by the Free Software Foundation, or (3) any later version of the GNU General Public License.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this library program; it should be in the file `COPYING`. If not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111 USA

For licensing inquiries, contact the author at:

```
Mark Jason Dominus
255 S. Warnock St.
Philadelphia, PA 19107
```

WARRANTY

`Tie::File` version 0.97 comes with ABSOLUTELY NO WARRANTY. For details, see the license.

THANKS

Gigantic thanks to Jarkko Hietaniemi, for agreeing to put this in the core when I hadn't written it yet, and for generally being helpful, supportive, and competent. (Usually the rule is "choose any one.") Also big thanks to Abhijit Menon-Sen for all of the same things.

Special thanks to Craig Berry and Peter Prymmer (for VMS portability help), Randy Kobes (for Win32 portability help), Clinton Pierce and Autrijus Tang (for heroic eleventh-hour Win32 testing above and beyond the call of duty), Michael G Schwern (for testing advice), and the rest of the CPAN testers (for testing generally).

Special thanks to Tels for suggesting several speed and memory optimizations.

Additional thanks to: Edward Avis / Mattia Barbon / Tom Christiansen / Gerrit Haase / Gurusamy Sarathy / Jarkko Hietaniemi (again) / Nikola Knezevic / John Kominetz / Nick Ing-Simmons / Tassilo von Parseval / H. Dieter Pearcey / Slaven Rezic / Eric Roode / Peter Scott / Peter Somu / Autrijus Tang (again) / Tels (again) / Juerd Waalboer

TODO

More tests. (Stuff I didn't think of yet.)

Paragraph mode?

Fixed-length mode. Leave-blanks mode.

Maybe an autolocking mode?

For many common uses of the module, the read cache is a liability. For example, a program that

inserts a single record, or that scans the file once, will have a cache hit rate of zero. This suggests a major optimization: The cache should be initially disabled. Here's a hybrid approach: Initially, the cache is disabled, but the cache code maintains statistics about how high the hit rate would be *if* it were enabled. When it sees the hit rate get high enough, it enables itself. The STAT comments in this code are the beginning of an implementation of this.

Record locking with fcntl()? Then the module might support an undo log and get real transactions. What a tour de force that would be.

Keeping track of the highest cached record. This would allow reads-in-a-row to skip the cache lookup faster (if reading from 1..N with empty cache at start, the last cached value will be always N-1).

More tests.