NAME

File::Temp - return name and handle of a temporary file safely

PORTABILITY

This section is at the top in order to provide easier access to porters. It is not expected to be rendered by a standard pod formatting tool. Please skip straight to the SYNOPSIS section if you are not trying to port this module to a new platform.

This module is designed to be portable across operating systems and it currently supports Unix, VMS, DOS, OS/2, Windows and Mac OS (Classic). When porting to a new OS there are generally three main issues that have to be solved:

- Can the OS unlink an open file? If it can not then the `_can_unlink_opened_file` method should be modified.
- Are the return values from `stat` reliable? By default all the return values from `stat` are compared when unlinking a temporary file using the filename and the handle. Operating systems other than unix do not always have valid entries in all fields. If utility function `File::Temp::unlink0` fails then the `stat` comparison should be modified accordingly.
- Security. Systems that can not support a test for the sticky bit on a directory can not use the MEDIUM and HIGH security tests. The `_can_do_level` method should be modified accordingly.

SYNOPSIS

```
use File::Temp qw/ tempfile tempdir /;

$fh = tempfile();
($fh, $filename) = tempfile();

($fh, $filename) = tempfile( $template, DIR => $dir);
($fh, $filename) = tempfile( $template, SUFFIX => '.dat');
($fh, $filename) = tempfile( $template, TMPDIR => 1 );

binmode( $fh, ":utf8" );

$dir = tempdir( CLEANUP => 1 );
($fh, $filename) = tempfile( DIR => $dir );
```

Object interface:

```
require File::Temp;
use File::Temp ();
use File::Temp qw/ :seekable /;

$fh = File::Temp->new();
$fname = $fh->filename;

$fh = File::Temp->new(TEMPLATE => $template);
$fname = $fh->filename;

$tmp = File::Temp->new( UNLINK => 0, SUFFIX => '.dat' );
print $tmp "Some data\n";
print "Filename is $tmp\n";
$tmp->seek( 0, SEEK_END );
```

The following interfaces are provided for compatibility with existing APIs. They should not be used in new code.

MkTemp family:

```
use File::Temp qw/ :mktemp  /;

($fh, $file) = mkstemp( "tmpfileXXXXX" );
($fh, $file) = mkstemps( "tmpfileXXXXXX", $suffix);

$tmpdir = mkdtemp( $template );

$unopened_file = mktemp( $template );
```

POSIX functions:

```
use File::Temp qw/ :POSIX /;

$file = tmpnam();
$fh = tmpfile();

($fh, $file) = tmpnam();
```

Compatibility functions:

```
$unopened_file = File::Temp::tempnam( $dir, $pfx );
```

DESCRIPTION

`File::Temp` can be used to create and open temporary files in a safe way. There is both a function interface and an object-oriented interface. The File::Temp constructor or the tempfile() function can be used to return the name and the open filehandle of a temporary file. The tempdir() function can be used to create a temporary directory.

The security aspect of temporary file creation is emphasized such that a filehandle and filename are returned together. This helps guarantee that a race condition can not occur where the temporary file is created by another process between checking for the existence of the file and its opening. Additional security levels are provided to check, for example, that the sticky bit is set on world writable directories. See *safe_level* for more information.

For compatibility with popular C library functions, Perl implementations of the mkstemp() family of functions are provided. These are, mkstemp(), mkstemps(), mkdtemp() and mktemp().

Additionally, implementations of the standard *POSIX* tmpnam() and tmpfile() functions are provided if required.

Implementations of mktemp(), tmpnam(), and tempnam() are provided, but should be used with caution since they return only a filename that was valid when function was called, so cannot guarantee that the file will not exist by the time the caller opens the filename.

Filehandles returned by these functions support the seekable methods.

OBJECT-ORIENTED INTERFACE

This is the primary interface for interacting with `File::Temp`. Using the OO interface a temporary file can be created when the object is constructed and the file can be removed when the object is no longer required.

Note that there is no method to obtain the filehandle from the `File::Temp` object. The object itself acts as a filehandle. The object isa `IO::Handle` and isa `IO::Seekable` so all those methods are available.

Also, the object is configured such that it stringifies to the name of the temporary file and so can be compared to a filename directly. It numifies to the `refaddr` the same as other handles and so can be compared to other handles with ==.

```
$fh eq $filename       # as a string
$fh != \*STDOUT        # as a number
```

new

Create a temporary file object.

```
my $tmp = File::Temp->new();
```

by default the object is constructed as if `tempfile` was called without options, but with the additional behaviour that the temporary file is removed by the object destructor if UNLINK is set to true (the default).

Supported arguments are the same as for `tempfile`: UNLINK (defaulting to true), DIR, EXLOCK and SUFFIX. Additionally, the filename template is specified using the TEMPLATE option. The OPEN option is not supported (the file is always opened).

```
$tmp = File::Temp->new( TEMPLATE => 'tempXXXXX',
                        DIR => 'mydir',
                        SUFFIX => '.dat');
```

Arguments are case insensitive.

Can call croak() if an error occurs.

newdir

Create a temporary directory using an object oriented interface.

```
$dir = File::Temp->newdir();
```

By default the directory is deleted when the object goes out of scope.

Supports the same options as the `tempdir` function. Note that directories created with this method default to CLEANUP => 1.

```
$dir = File::Temp->newdir( $template, %options );
```

A template may be specified either with a leading template or with a TEMPLATE argument.

filename

Return the name of the temporary file associated with this object (if the object was created using the "new" constructor).

```
$filename = $tmp->filename;
```

This method is called automatically when the object is used as a string.

dirname

Return the name of the temporary directory associated with this object (if the object was created using the "newdir" constructor).

```
$dirname = $tmpdir->dirname;
```

This method is called automatically when the object is used in string context.

unlink_on_destroy

Control whether the file is unlinked when the object goes out of scope. The file is removed if this value is true and $KEEP_ALL is not.

```
 $fh->unlink_on_destroy( 1 );
```

Default is for the file to be removed.

DESTROY

When the object goes out of scope, the destructor is called. This destructor will attempt to unlink the file (using *unlink1*) if the constructor was called with UNLINK set to 1 (the default state if UNLINK is not specified).

No error is given if the unlink fails.

If the object has been passed to a child process during a fork, the file will be deleted when the object goes out of scope in the parent.

For a temporary directory object the directory will be removed unless the CLEANUP argument was used in the constructor (and set to false) or `unlink_on_destroy` was modified after creation. Note that if a temp directory is your current directory, it cannot be removed - a warning will be given in this case. `chdir()` out of the directory before letting the object go out of scope.

If the global variable $KEEP_ALL is true, the file or directory will not be removed.

FUNCTIONS

This section describes the recommended interface for generating temporary files and directories.

tempfile

This is the basic function to generate temporary files. The behaviour of the file can be changed using various options:

```
  $fh = tempfile();
  ($fh, $filename) = tempfile();
```

Create a temporary file in the directory specified for temporary files, as specified by the tmpdir() function in *File::Spec*.

```
  ($fh, $filename) = tempfile($template);
```

Create a temporary file in the current directory using the supplied template. Trailing `X' characters are replaced with random letters to generate the filename. At least four `X' characters must be present at the end of the template.

```
  ($fh, $filename) = tempfile($template, SUFFIX => $suffix)
```

Same as previously, except that a suffix is added to the template after the `X' translation. Useful for ensuring that a temporary filename has a particular extension when needed by other applications. But see the WARNING at the end.

```
  ($fh, $filename) = tempfile($template, DIR => $dir);
```

Translates the template as before except that a directory name is specified.

```
  ($fh, $filename) = tempfile($template, TMPDIR => 1);
```

Equivalent to specifying a DIR of "File::Spec->tmpdir", writing the file into the same temporary directory as would be used if no template was specified at all.

```
  ($fh, $filename) = tempfile($template, UNLINK => 1);
```

Return the filename and filehandle as before except that the file is automatically removed

when the program exits (dependent on $KEEP_ALL). Default is for the file to be removed if a file handle is requested and to be kept if the filename is requested. In a scalar context (where no filename is returned) the file is always deleted either (depending on the operating system) on exit or when it is closed (unless $KEEP_ALL is true when the temp file is created).

Use the object-oriented interface if fine-grained control of when a file is removed is required.

If the template is not specified, a template is always automatically generated. This temporary file is placed in tmpdir() (*File::Spec*) unless a directory is specified explicitly with the DIR option.

```
$fh = tempfile( DIR => $dir );
```

If called in scalar context, only the filehandle is returned and the file will automatically be deleted when closed on operating systems that support this (see the description of tmpfile() elsewhere in this document). This is the preferred mode of operation, as if you only have a filehandle, you can never create a race condition by fumbling with the filename. On systems that can not unlink an open file or can not mark a file as temporary when it is opened (for example, Windows NT uses the `O_TEMPORARY` flag) the file is marked for deletion when the program ends (equivalent to setting UNLINK to 1). The `UNLINK` flag is ignored if present.

```
(undef, $filename) = tempfile($template, OPEN => 0);
```

This will return the filename based on the template but will not open this file. Cannot be used in conjunction with UNLINK set to true. Default is to always open the file to protect from possible race conditions. A warning is issued if warnings are turned on. Consider using the tmpnam() and mktemp() functions described elsewhere in this document if opening the file is not required.

If the operating system supports it (for example BSD derived systems), the filehandle will be opened with O_EXLOCK (open with exclusive file lock). This can sometimes cause problems if the intention is to pass the filename to another system that expects to take an exclusive lock itself (such as DBD::SQLite) whilst ensuring that the tempfile is not reused. In this situation the "EXLOCK" option can be passed to tempfile. By default EXLOCK will be true (this retains compatibility with earlier releases).

```
($fh, $filename) = tempfile($template, EXLOCK => 0);
```

Options can be combined as required.

Will croak() if there is an error.

tempdir

This is the recommended interface for creation of temporary directories. By default the directory will not be removed on exit (that is, it won't be temporary; this behaviour can not be changed because of issues with backwards compatibility). To enable removal either use the CLEANUP option which will trigger removal on program exit, or consider using the "newdir" method in the object interface which will allow the directory to be cleaned up when the object goes out of scope.

The behaviour of the function depends on the arguments:

```
$tempdir = tempdir();
```

Create a directory in tmpdir() (see *File::Spec*).

```
$tempdir = tempdir( $template );
```

Create a directory from the supplied template. This template is similar to that described for tempfile(). `X' characters at the end of the template are replaced with random letters to construct the directory name. At least four `X' characters must be in the template.

```
$tempdir = tempdir ( DIR => $dir );
```

Specifies the directory to use for the temporary directory. The temporary directory name is derived from an internal template.

```
$tempdir = tempdir ( $template, DIR => $dir );
```

Prepend the supplied directory name to the template. The template should not include parent directory specifications itself. Any parent directory specifications are removed from the template before prepending the supplied directory.

```
$tempdir = tempdir ( $template, TMPDIR => 1 );
```

Using the supplied template, create the temporary directory in a standard location for temporary files. Equivalent to doing

```
$tempdir = tempdir ( $template, DIR => File::Spec->tmpdir);
```

but shorter. Parent directory specifications are stripped from the template itself. The `TMPDIR` option is ignored if `DIR` is set explicitly. Additionally, `TMPDIR` is implied if neither a template nor a directory are supplied.

```
$tempdir = tempdir( $template, CLEANUP => 1);
```

Create a temporary directory using the supplied template, but attempt to remove it (and all files inside it) when the program exits. Note that an attempt will be made to remove all files from the directory even if they were not created by this module (otherwise why ask to clean it up?). The directory removal is made with the rmtree() function from the *File::Path* module. Of course, if the template is not specified, the temporary directory will be created in tmpdir() and will also be removed at program exit.

Will croak() if there is an error.

MKTEMP FUNCTIONS

The following functions are Perl implementations of the mktemp() family of temp file generation system calls.

mkstemp

Given a template, returns a filehandle to the temporary file and the name of the file.

```
($fh, $name) = mkstemp( $template );
```

In scalar context, just the filehandle is returned.

The template may be any filename with some number of X's appended to it, for example */tmp/temp.XXXX*. The trailing X's are replaced with unique alphanumeric combinations.

Will croak() if there is an error.

mkstemps

Similar to mkstemp(), except that an extra argument can be supplied with a suffix to be appended to the template.

```
($fh, $name) = mkstemps( $template, $suffix );
```

For example a template of `testXXXXXX` and suffix of `.dat` would generate a file similar to *testhGji_w.dat*.

Returns just the filehandle alone when called in scalar context.

Will croak() if there is an error.

mkdtemp

Create a directory from a template. The template must end in X's that are replaced by the routine.

```
$tmpdir_name = mkdtemp($template);
```

Returns the name of the temporary directory created.

Directory must be removed by the caller.

Will croak() if there is an error.

mktemp

Returns a valid temporary filename but does not guarantee that the file will not be opened by someone else.

```
$unopened_file = mktemp($template);
```

Template is the same as that required by mkstemp().

Will croak() if there is an error.

POSIX FUNCTIONS

This section describes the re-implementation of the tmpnam() and tmpfile() functions described in *POSIX* using the mkstemp() from this module.

Unlike the *POSIX* implementations, the directory used for the temporary file is not specified in a system include file (`P_tmpdir`) but simply depends on the choice of tmpdir() returned by *File::Spec*. On some implementations this location can be set using the `TMPDIR` environment variable, which may not be secure. If this is a problem, simply use mkstemp() and specify a template.

tmpnam

When called in scalar context, returns the full name (including path) of a temporary file (uses mktemp()). The only check is that the file does not already exist, but there is no guarantee that that condition will continue to apply.

```
$file = tmpnam();
```

When called in list context, a filehandle to the open file and a filename are returned. This is achieved by calling mkstemp() after constructing a suitable template.

```
($fh, $file) = tmpnam();
```

If possible, this form should be used to prevent possible race conditions.

See *"tmpdir" in File::Spec* for information on the choice of temporary directory for a particular operating system.

Will croak() if there is an error.

tmpfile

Returns the filehandle of a temporary file.

```
$fh = tmpfile();
```

The file is removed when the filehandle is closed or when the program exits. No access to the filename is provided.

If the temporary file can not be created undef is returned. Currently this command will probably not work when the temporary directory is on an NFS file system.

Will croak() if there is an error.

ADDITIONAL FUNCTIONS

These functions are provided for backwards compatibility with common tempfile generation C library functions.

They are not exported and must be addressed using the full package name.

tempnam

Return the name of a temporary file in the specified directory using a prefix. The file is guaranteed not to exist at the time the function was called, but such guarantees are good for one clock tick only. Always use the proper form of `sysopen` with `O_CREAT | O_EXCL` if you must open such a filename.

```
$filename = File::Temp::tempnam( $dir, $prefix );
```

Equivalent to running mktemp() with $dir/$prefixXXXXXXXX (using unix file convention as an example)

Because this function uses mktemp(), it can suffer from race conditions.

Will croak() if there is an error.

UTILITY FUNCTIONS

Useful functions for dealing with the filehandle and filename.

unlink0

Given an open filehandle and the associated filename, make a safe unlink. This is achieved by first checking that the filename and filehandle initially point to the same file and that the number of links to the file is 1 (all fields returned by stat() are compared). Then the filename is unlinked and the filehandle checked once again to verify that the number of links on that file is now 0. This is the closest you can come to making sure that the filename unlinked was the same as the file whose descriptor you hold.

```
unlink0($fh, $path)
   or die "Error unlinking file $path safely";
```

Returns false on error but croaks() if there is a security anomaly. The filehandle is not closed since on some occasions this is not required.

On some platforms, for example Windows NT, it is not possible to unlink an open file (the file must be closed first). On those platforms, the actual unlinking is deferred until the program ends and good status is returned. A check is still performed to make sure that the filehandle and filename are pointing to the same thing (but not at the time the end block is executed since the deferred removal may not have access to the filehandle).

Additionally, on Windows NT not all the fields returned by stat() can be compared. For example, the `dev` and `rdev` fields seem to be different. Also, it seems that the size of the file returned by stat() does not always agree, with `stat(FH)` being more accurate than `stat(filename)`, presumably because of caching issues even when using autoflush (this is usually overcome by waiting a while after writing to the tempfile before attempting to `unlink0` it).

Finally, on NFS file systems the link count of the file handle does not always go to zero immediately after unlinking. Currently, this command is expected to fail on NFS disks.

This function is disabled if the global variable $KEEP_ALL is true and an unlink on open file is supported. If the unlink is to be deferred to the END block, the file is still registered for removal.

This function should not be called if you are using the object oriented interface since the it will interfere with the object destructor deleting the file.

cmpstat

Compare `stat` of filehandle with `stat` of provided filename. This can be used to check that the filename and filehandle initially point to the same file and that the number of links to the file is 1 (all fields returned by stat() are compared).

```
cmpstat($fh, $path)
   or die "Error comparing handle with file";
```

Returns false if the stat information differs or if the link count is greater than 1. Calls croak if there is a security anomaly.

On certain platforms, for example Windows, not all the fields returned by stat() can be compared. For example, the `dev` and `rdev` fields seem to be different in Windows. Also, it seems that the size of the file returned by stat() does not always agree, with `stat(FH)` being more accurate than `stat(filename)`, presumably because of caching issues even when using autoflush (this is usually overcome by waiting a while after writing to the tempfile before attempting to `unlink0` it).

Not exported by default.

unlink1

Similar to `unlink0` except after file comparison using cmpstat, the filehandle is closed prior to attempting to unlink the file. This allows the file to be removed without using an END block, but does mean that the post-unlink comparison of the filehandle state provided by `unlink0` is not available.

```
unlink1($fh, $path)
   or die "Error closing and unlinking file";
```

Usually called from the object destructor when using the OO interface.

Not exported by default.

This function is disabled if the global variable $KEEP_ALL is true.

Can call croak() if there is a security anomaly during the stat() comparison.

cleanup

Calling this function will cause any temp files or temp directories that are registered for removal to be removed. This happens automatically when the process exits but can be triggered manually if the caller is sure that none of the temp files are required. This method can be registered as an Apache callback.

Note that if a temp directory is your current directory, it cannot be removed. `chdir()` out of the directory first before calling `cleanup()`. (For the cleanup at program exit when the CLEANUP flag is set, this happens automatically.)

On OSes where temp files are automatically removed when the temp file is closed, calling this function will have no effect other than to remove temporary directories (which may include temporary files).

```
File::Temp::cleanup();
```

Not exported by default.

PACKAGE VARIABLES

These functions control the global state of the package.

safe_level

Controls the lengths to which the module will go to check the safety of the temporary file or directory before proceeding. Options are:

STANDARD

Do the basic security measures to ensure the directory exists and is writable, that temporary files are opened only if they do not already exist, and that possible race conditions are avoided. Finally the *unlink0* function is used to remove files safely.

MEDIUM

In addition to the STANDARD security, the output directory is checked to make

sure that it is owned either by root or the user running the program. If the directory is writable by group or by other, it is then checked to make sure that the sticky bit is set.

Will not work on platforms that do not support the `-k` test for sticky bit.

HIGH

In addition to the MEDIUM security checks, also check for the possibility of ``chown() giveaway'' using the *POSIX* sysconf() function. If this is a possibility, each directory in the path is checked in turn for safeness, recursively walking back to the root directory.

For platforms that do not support the *POSIX* `_PC_CHOWN_RESTRICTED` symbol (for example, Windows NT) it is assumed that ``chown() giveaway'' is possible and the recursive test is performed.

The level can be changed as follows:

```
File::Temp->safe_level( File::Temp::HIGH );
```

The level constants are not exported by the module.

Currently, you must be running at least perl v5.6.0 in order to run with MEDIUM or HIGH security. This is simply because the safety tests use functions from *Fcntl* that are not available in older versions of perl. The problem is that the version number for Fcntl is the same in perl 5.6.0 and in 5.005_03 even though they are different versions.

On systems that do not support the HIGH or MEDIUM safety levels (for example Win NT or OS/2) any attempt to change the level will be ignored. The decision to ignore rather than raise an exception allows portable programs to be written with high security in mind for the systems that can support this without those programs failing on systems where the extra tests are irrelevant.

If you really need to see whether the change has been accepted simply examine the return value of `safe_level`.

```
$newlevel = File::Temp->safe_level( File::Temp::HIGH );
die "Could not change to high security"
    if $newlevel != File::Temp::HIGH;
```

TopSystemUID

This is the highest UID on the current system that refers to a root UID. This is used to make sure that the temporary directory is owned by a system UID (`root`, `bin`, `sys` etc) rather than simply by root.

This is required since on many unix systems `/tmp` is not owned by root.

Default is to assume that any UID less than or equal to 10 is a root UID.

```
File::Temp->top_system_uid(10);
my $topid = File::Temp->top_system_uid;
```

This value can be adjusted to reduce security checking if required. The value is only relevant when `safe_level` is set to MEDIUM or higher.

$KEEP_ALL

Controls whether temporary files and directories should be retained regardless of any instructions in the program to remove them automatically. This is useful for debugging but should not be used in production code.

```
$File::Temp::KEEP_ALL = 1;
```

Default is for files to be removed as requested by the caller.

In some cases, files will only be retained if this variable is true when the file is created. This means that you can not create a temporary file, set this variable and expect the temp file to still be around when the program exits.

$DEBUG

Controls whether debugging messages should be enabled.

```
$File::Temp::DEBUG = 1;
```

Default is for debugging mode to be disabled.

WARNING

For maximum security, endeavour always to avoid ever looking at, touching, or even imputing the existence of the filename. You do not know that that filename is connected to the same file as the handle you have, and attempts to check this can only trigger more race conditions. It's far more secure to use the filehandle alone and dispense with the filename altogether.

If you need to pass the handle to something that expects a filename then on a unix system you can use `"/dev/fd/" . fileno($fh)` for arbitrary programs. Perl code that uses the 2-argument version of `open` can be passed `"+<=&" . fileno($fh)`. Otherwise you will need to pass the filename. You will have to clear the close-on-exec bit on that file descriptor before passing it to another process.

```
use Fcntl qw/F_SETFD F_GETFD/;
fcntl($tmpfh, F_SETFD, 0)
    or die "Can't clear close-on-exec flag on temp fh: $!\n";
```

Temporary files and NFS

Some problems are associated with using temporary files that reside on NFS file systems and it is recommended that a local filesystem is used whenever possible. Some of the security tests will most probably fail when the temp file is not local. Additionally, be aware that the performance of I/O operations over NFS will not be as good as for a local disk.

Forking

In some cases files created by File::Temp are removed from within an END block. Since END blocks are triggered when a child process exits (unless `POSIX::_exit()` is used by the child) File::Temp takes care to only remove those temp files created by a particular process ID. This means that a child will not attempt to remove temp files created by the parent process.

If you are forking many processes in parallel that are all creating temporary files, you may need to reset the random number seed using srand(EXPR) in each child else all the children will attempt to walk through the same set of random file names and may well cause themselves to give up if they exceed the number of retry attempts.

Directory removal

Note that if you have chdir'ed into the temporary directory and it is subsequently cleaned up (either in the END block or as part of object destruction), then you will get a warning from File::Path::rmtree().

Taint mode

If you need to run code under taint mode, updating to the latest *File::Spec* is highly recommended.

BINMODE

The file returned by File::Temp will have been opened in binary mode if such a mode is available. If that is not correct, use the `binmode()` function to change the mode of the filehandle.

Note that you can modify the encoding of a file opened by File::Temp also by using `binmode()`.

HISTORY

Originally began life in May 1999 as an XS interface to the system mkstemp() function. In March 2000, the OpenBSD mkstemp() code was translated to Perl for total control of the code's security checking, to ensure the presence of the function regardless of operating system and to help with portability. The module was shipped as a standard part of perl from v5.6.1.

SEE ALSO

"tmpnam" in POSIX, *"tmpfile" in POSIX*, *File::Spec*, *File::Path*

See *IO::File* and *File::MkTemp*, *Apache::TempFile* for different implementations of temporary file handling.

See *File::Tempdir* for an alternative object-oriented wrapper for the `tempdir` function.

AUTHOR

Tim Jenness <tjenness@cpan.org>

Original Perl implementation loosely based on the OpenBSD C code for mkstemp(). Thanks to Tom Christiansen for suggesting that this module should be written and providing ideas for code improvements and security enhancements.