NAME

"Test::Refcount" – assert reference counts on objects

SYNOPSIS

```
use Test::More tests => 2;
use Test::Refcount;

use Some::Class;

my $object = Some::Class->new();

is_oneref( $object, '$object has a refcount of 1' );

my $otherref = $object;

is_refcount( $object, 2, '$object now has 2 references' );
```

DESCRIPTION

The Perl garbage collector uses simple reference counting during the normal execution of a program. This means that cycles or unweakened references in other parts of code can keep an object around for longer than intended. To help avoid this problem, the reference count of a new object from its class constructor ought to be 1. This way, the caller can know the object will be properly DESTROYed when it drops all of its references to it.

This module provides two test functions to help ensure this property holds for an object class, so as to be polite to its callers.

If the assertion fails; that is, if the actual reference count is different to what was expected, either of the following two modules may be used to assist the developer in finding where the references are.

- If Devel::MAT is installed, this test module will use it to dump the state of the memory after a failure. It will create a *.pmat* file named the same as the unit test, but with the trailing *.t* suffix replaced with *–TEST.pmat* where `TEST` is the number of the test that failed (in case there was more than one).
- If Devel::FindRef module is installed, a reverse-references trace is printed to the test output.

See the examples below for more information.

FUNCTIONS

is_refcount

```
is_refcount( $object, $count, $name )
```

Test that `$object` has `$count` references to it.

is_oneref

```
is_oneref( $object, $name )
```

Assert that the `$object` has only 1 reference to it.

refcount

```
$count = refcount( $object )
```

Since version 0.09.

Returns the reference count of the given object as used by the test functions. This is useful for making tests that don't care what the count is before they start, but simply assert that the count hasn't changed by the end.

```
use Test::Refcount import => [qw( is_refcount refcount )];
{
   my $count = refcount( $object );

   do_something( $object );
```

```
        is_refcount( $object, $count, 'do_something() preserves refcount' );
    }
```

EXAMPLE

Suppose, having written a new class `MyBall`, you now want to check that its constructor and methods are well-behaved, and don't leak references. Consider the following test script:

```
use Test::More tests => 2;
use Test::Refcount;

use MyBall;

my $ball = MyBall->new();
is_oneref( $ball, 'One reference after construct' );

$ball->bounce;

# Any other code here that might be part of the test script

is_oneref( $ball, 'One reference just before EOF' );
```

The first assertion is just after the constructor, to check that the reference returned by it is the only reference to that object. This fact is important if we ever want `DESTROY` to behave properly. The second call is right at the end of the file, just before the main scope closes. At this stage we expect the reference count also to be one, so that the object is properly cleaned up.

Suppose, when run, this produces the following output (presuming Devel::MAT::Dumper is available):

```
1..2
ok 1 - One reference after construct
not ok 2 - One reference just before EOF
#   Failed test 'One reference just before EOF'
#   at ex.pl line 26.
#   expected 1 references, found 2
# SV address is 0x55e14c310278
# Writing heap dump to ex-2.pmat
# Looks like you failed 1 test of 2.
```

This has written a *ex−2.pmat* file we can load using the `pmat` shell and use the `identify` command on the given address to find where it went:

```
$ pmat ex-2.pmat
Perl memory dumpfile from perl 5.28.1 threaded
Heap contains 25233 objects
pmat> identify 0x55e14c310278
HASH(0)=MyBall at 0x55e14c310278 is:
(via RV) the lexical $ball at depth 1 of CODE() at 0x55e14c3104a0=main_cv, which
│ the main code
(via RV) value {self} of HASH(2) at 0x55e14cacb860, which is (*A):
  (via RV) value {cycle} of HASH(2) at 0x55e14cacb860, which is:
    itself
```

(This document isn't intended to be a full tutorial on Devel::MAT and the `pmat` shell; for that see Devel::MAT::UserGuide).

Alternatively, this produces the following output when using Devel::FindRef instead:

```
1..2
ok 1 - One reference after construct
not ok 2 - One reference just before EOF
#   Failed test 'One reference just before EOF'
#   at demo.pl line 16.
#   expected 1 references, found 2
# MyBall=ARRAY(0x817f880) is
# +- referenced by REF(0x82c1fd8), which is
# |     in the member 'self' of HASH(0x82c1f68), which is
# |        referenced by REF(0x81989d0), which is
# |           in the member 'cycle' of HASH(0x82c1f68), which was seen before.
# +- referenced by REF(0x82811d0), which is
#       in the lexical '$ball' in CODE(0x817fa00), which is
#          the main body of the program.
# Looks like you failed 1 test of 2.
```

From this output, we can see that the constructor was well-behaved, but that a reference was leaked by the end of the script – the reference count was 2, when we expected just 1. Reading the trace output, we can see that there were 2 references that could be found – one stored in the `$ball` lexical in the main program, and one stored in a HASH. Since we expected to find the `$ball` lexical variable, we know we are now looking for a leak in a hash somewhere in the code. From reading the test script, we can guess this leak is likely to be in the **bounce()** method. Furthermore, we know that the reference to the object will be stored in a HASH in a member called `self`.

By reading the code which implements the **bounce()** method, we can see this is indeed the case:

```
sub bounce
{
   my $self = shift;
   my $cycle = { self => $self };
   $cycle->{cycle} = $cycle;
}
```

From reading the tracing output, we find that the HASH this object is referenced in also contains a reference to itself, in a member called `cycle`. This comes from the last line in this function, a line that purposely created a cycle, to demonstrate the point. While a real program probably wouldn't do anything quite this obvious, the trace would still be useful in finding the likely cause of the leak.

If neither `Devel::MAT::Dumper` nor `Devel::FindRef` are available, then these detailed traces will not be produced. The basic reference count testing will still take place, but a smaller message will be produced:

```
1..2
ok 1 - One reference after construct
not ok 2 - One reference just before EOF
#   Failed test 'One reference just before EOF'
#   at demo.pl line 16.
#   expected 1 references, found 2
# Looks like you failed 1 test of 2.
```

BUGS

- Temporaries created on the stack

  Code which creates temporaries on the stack, to be released again when the called function returns does not work correctly on perl 5.8 (and probably before). Examples such as

  ```
  is_oneref( [] );
  ```

  may fail and claim a reference count of 2 instead.

  Passing a variable such as

```
    my $array = [];
    is_oneref( $array );
```

works fine. Because of the intention of this test module; that is, to assert reference counts on some object stored in a variable during the lifetime of the test script, this is unlikely to cause any problems.

ACKNOWLEDGEMENTS

Peter Rabbitson <ribasushi@cpan.org> – for suggesting using core's `B` instead of `Devel::Refcount` to obtain refcounts

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>