NAME

Memoize - Make functions faster by trading space for time

SYNOPSIS

```
       # This is the documentation for Memoize 1.01
use Memoize;
memoize('slow_function');
slow_function(arguments);    # Is faster than it was before
```

This is normally all you need to know. However, many options are available:

```
memoize(function, options...);
```

Options include:

```
NORMALIZER => function
INSTALL => new_name

SCALAR_CACHE => 'MEMORY'
       SCALAR_CACHE => ['HASH', \%cache_hash ]
SCALAR_CACHE => 'FAULT'
SCALAR_CACHE => 'MERGE'

LIST_CACHE => 'MEMORY'
       LIST_CACHE => ['HASH', \%cache_hash ]
LIST_CACHE => 'FAULT'
LIST_CACHE => 'MERGE'
```

DESCRIPTION

`Memoizing' a function makes it faster by trading space for time. It does this by caching the return values of the function in a table. If you call the function again with the same arguments, `memoize` jumps in and gives you the value out of the table, instead of letting the function compute the value all over again.

Here is an extreme example. Consider the Fibonacci sequence, defined by the following function:

```
# Compute Fibonacci numbers
sub fib {
  my $n = shift;
  return $n if $n < 2;
  fib($n-1) + fib($n-2);
}
```

This function is very slow. Why? To compute fib(14), it first wants to compute fib(13) and fib(12), and add the results. But to compute fib(13), it first has to compute fib(12) and fib(11), and then it comes back and computes fib(12) all over again even though the answer is the same. And both of the times that it wants to compute fib(12), it has to compute fib(11) from scratch, and then it has to do it again each time it wants to compute fib(13). This function does so much recomputing of old results that it takes a really long time to run---fib(14) makes 1,200 extra recursive calls to itself, to compute and recompute things that it already computed.

This function is a good candidate for memoization. If you memoize the `fib' function above, it will compute fib(14) exactly once, the first time it needs to, and then save the result in a table. Then if you ask for fib(14) again, it gives you the result out of the table. While computing fib(14), instead of computing fib(12) twice, it does it once; the second time it needs the value it gets it from the table. It

doesn't compute fib(11) four times; it computes it once, getting it from the table the next three times. Instead of making 1,200 recursive calls to `fib', it makes 15. This makes the function about 150 times faster.

You could do the memoization yourself, by rewriting the function, like this:

```
# Compute Fibonacci numbers, memoized version
{ my @fib;
    sub fib {
    my $n = shift;
    return $fib[$n] if defined $fib[$n];
    return $fib[$n] = $n if $n < 2;
    $fib[$n] = fib($n-1) + fib($n-2);
  }
      }
```

Or you could use this module, like this:

```
use Memoize;
memoize('fib');

# Rest of the fib function just like the original version.
```

This makes it easy to turn memoizing on and off.

Here's an even simpler example: I wrote a simple ray tracer; the program would look in a certain direction, figure out what it was looking at, and then convert the `color' value (typically a string like `red') of that object to a red, green, and blue pixel value, like this:

```
    for ($direction = 0; $direction < 300; $direction++) {
      # Figure out which object is in direction $direction
      $color = $object->{color};
      ($r, $g, $b) = @{&ColorToRGB($color)};
      ...
    }
```

Since there are relatively few objects in a picture, there are only a few colors, which get looked up over and over again. Memoizing `ColorToRGB` sped up the program by several percent.

DETAILS

This module exports exactly one function, `memoize`. The rest of the functions in this package are None of Your Business.

You should say

```
memoize(function)
```

where `function` is the name of the function you want to memoize, or a reference to it. `memoize` returns a reference to the new, memoized version of the function, or `undef` on a non-fatal error. At present, there are no non-fatal errors, but there might be some in the future.

If `function` was the name of a function, then `memoize` hides the old version and installs the new memoized version under the old name, so that `&function(...)` actually invokes the memoized version.

OPTIONS

There are some optional options you can pass to `memoize` to change the way it behaves a little. To supply options, invoke `memoize` like this:

```
memoize(function, NORMALIZER => function,
    INSTALL => newname,
                         SCALAR_CACHE => option,
                LIST_CACHE => option
   );
```

Each of these options is optional; you can include some, all, or none of them.

INSTALL

If you supply a function name with `INSTALL`, memoize will install the new, memoized version of the function under the name you give. For example,

```
memoize('fib', INSTALL => 'fastfib')
```

installs the memoized version of `fib` as `fastfib`; without the `INSTALL` option it would have replaced the old `fib` with the memoized version.

To prevent `memoize` from installing the memoized version anywhere, use `INSTALL => undef`.

NORMALIZER

Suppose your function looks like this:

```
# Typical call: f('aha!', A => 11, B => 12);
sub f {
  my $a = shift;
  my %hash = @_;
  $hash{B} ||= 2;  # B defaults to 2
  $hash{C} ||= 7;  # C defaults to 7

  # Do something with $a, %hash
}
```

Now, the following calls to your function are all completely equivalent:

```
f(OUCH);
f(OUCH, B => 2);
f(OUCH, C => 7);
f(OUCH, B => 2, C => 7);
f(OUCH, C => 7, B => 2);
(etc.)
```

However, unless you tell `Memoize` that these calls are equivalent, it will not know that, and it will compute the values for these invocations of your function separately, and store them separately.

To prevent this, supply a `NORMALIZER` function that turns the program arguments into a string in a way that equivalent arguments turn into the same string. A `NORMALIZER` function for `f` above might look like this:

```
sub normalize_f {
  my $a = shift;
  my %hash = @_;
  $hash{B} ||= 2;
```

```
  $hash{C} ||= 7;

  join(',', $a, map ($_ => $hash{$_}) sort keys %hash);
}
```

Each of the argument lists above comes out of the `normalize_f` function looking exactly the same, like this:

```
OUCH,B,2,C,7
```

You would tell `Memoize` to use this normalizer this way:

```
memoize('f', NORMALIZER => 'normalize_f');
```

`memoize` knows that if the normalized version of the arguments is the same for two argument lists, then it can safely look up the value that it computed for one argument list and return it as the result of calling the function with the other argument list, even if the argument lists look different.

The default normalizer just concatenates the arguments with character 28 in between. (In ASCII, this is called FS or control-\.) This always works correctly for functions with only one string argument, and also when the arguments never contain character 28. However, it can confuse certain argument lists:

```
normalizer("a\034", "b")
normalizer("a", "\034b")
normalizer("a\034\034b")
```

for example.

Since hash keys are strings, the default normalizer will not distinguish between `undef` and the empty string. It also won't work when the function's arguments are references. For example, consider a function g which gets two arguments: A number, and a reference to an array of numbers:

```
g(13, [1,2,3,4,5,6,7]);
```

The default normalizer will turn this into something like `"13\034ARRAY(0x436c1f)"`. That would be all right, except that a subsequent array of numbers might be stored at a different location even though it contains the same data. If this happens, `Memoize` will think that the arguments are different, even though they are equivalent. In this case, a normalizer like this is appropriate:

```
sub normalize { join ' ', $_[0], @{$_[1]} }
```

For the example above, this produces the key "13 1 2 3 4 5 6 7".

Another use for normalizers is when the function depends on data other than those in its arguments. Suppose you have a function which returns a value which depends on the current hour of the day:

```
sub on_duty {
         my ($problem_type) = @_;
  my $hour = (localtime)[2];
         open my $fh, "$DIR/$problem_type" or die...;
         my $line;
         while ($hour-- > 0)
           $line = <$fh>;
         }
  return $line;
}
```

At 10:23, this function generates the 10th line of a data file; at 3:45 PM it generates the 15th line instead. By default, Memoize will only see the $problem_type argument. To fix this, include the current hour in the normalizer:

```
sub normalize { join ' ', (localtime)[2], @_ }
```

The calling context of the function (scalar or list context) is propagated to the normalizer. This means that if the memoized function will treat its arguments differently in list context than it would in scalar context, you can have the normalizer function select its behavior based on the results of wantarray. Even if called in a list context, a normalizer should still return a single string.

SCALAR_CACHE, LIST_CACHE

Normally, Memoize caches your function's return values into an ordinary Perl hash variable. However, you might like to have the values cached on the disk, so that they persist from one run of your program to the next, or you might like to associate some other interesting semantics with the cached values.

There's a slight complication under the hood of Memoize: There are actually *two* caches, one for scalar values and one for list values. When your function is called in scalar context, its return value is cached in one hash, and when your function is called in list context, its value is cached in the other hash. You can control the caching behavior of both contexts independently with these options.

The argument to LIST_CACHE or SCALAR_CACHE must either be one of the following four strings:

```
MEMORY
FAULT
MERGE
        HASH
```

or else it must be a reference to a list whose first element is one of these four strings, such as [HASH, arguments...].

MEMORY

MEMORY means that return values from the function will be cached in an ordinary Perl hash variable. The hash variable will not persist after the program exits. This is the default.

HASH

HASH allows you to specify that a particular hash that you supply will be used as the cache. You can tie this hash beforehand to give it any behavior you want.

A tied hash can have any semantics at all. It is typically tied to an on-disk database, so that cached values are stored in the database and retrieved from it again when needed, and the disk file typically persists after your program has exited. See perltie for more complete details about tie.

A typical example is:

```
use DB_File;
tie my %cache => 'DB_File', $filename, O_RDWR|O_CREAT, 0666;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

This has the effect of storing the cache in a DB_File database whose name is in $filename. The cache will persist after the program has exited. Next time the program runs, it will find the cache already populated from the previous run of the program. Or you can forcibly populate the cache by constructing a batch program that runs in the background and populates the cache file. Then when you come to run your real program the memoized function will be fast because all its results have been precomputed.

TIE

This option is no longer supported. It is still documented only to aid in the debugging of old programs that use it. Old programs should be converted to use the HASH option instead.

```
        memoize ... [TIE, PACKAGE, ARGS...]
```

is merely a shortcut for

```
        require PACKAGE;
 { my %cache;
          tie %cache, PACKAGE, ARGS...;
 }
        memoize ... [HASH => \%cache];
```

FAULT

FAULT means that you never expect to call the function in scalar (or list) context, and that if Memoize detects such a call, it should abort the program. The error message is one of

```
 `foo' function called in forbidden list context at line ...
 `foo' function called in forbidden scalar context at line ...
```

MERGE

MERGE normally means the function does not distinguish between list and sclar context, and that return values in both contexts should be stored together. LIST_CACHE => MERGE means that list context return values should be stored in the same hash that is used for scalar context returns, and SCALAR_CACHE => MERGE means the same, mutatis mutandis. It is an error to specify MERGE for both, but it probably does something useful.

Consider this function:

```
 sub pi { 3; }
```

Normally, the following code will result in two calls to pi:

```
    $x = pi();
    ($y) = pi();
    $z = pi();
```

The first call caches the value 3 in the scalar cache; the second caches the list (3) in the list cache. The third call doesn't call the real pi function; it gets the value from the scalar cache.

Obviously, the second call to pi is a waste of time, and storing its return value is a waste of space. Specifying LIST_CACHE => MERGE will make memoize use the same cache for scalar and list context return values, so that the second call uses the scalar cache that was populated by the first call. pi ends up being called only once, and both subsequent calls return 3 from the cache, regardless of the calling context.

Another use for MERGE is when you want both kinds of return values stored in the same disk file; this saves you from having to deal with two disk files instead of one. You can use a normalizer function to keep the two sets of return values separate. For example:

```
        tie my %cache => 'MLDBM', 'DB_File', $filename, ...;

 memoize 'myfunc',
   NORMALIZER => 'n',
   SCALAR_CACHE => [HASH => \%cache],
   LIST_CACHE => MERGE,
 ;

 sub n {
   my $context = wantarray() ? 'L' : 'S';
   # ... now compute the hash key from the arguments ...
```

```
  $hashkey = "$context:$hashkey";
}
```

This normalizer function will store scalar context return values in the disk file under keys that begin with `S:`, and list context return values under keys that begin with `L:`.

OTHER FACILITIES

unmemoize

There's an `unmemoize` function that you can import if you want to. Why would you want to? Here's an example: Suppose you have your cache tied to a DBM file, and you want to make sure that the cache is written out to disk if someone interrupts the program. If the program exits normally, this will happen anyway, but if someone types control-C or something then the program will terminate immediately without synchronizing the database. So what you can do instead is

```
$SIG{INT} = sub { unmemoize 'function' };
```

`unmemoize` accepts a reference to, or the name of a previously memoized function, and undoes whatever it did to provide the memoized version in the first place, including making the name refer to the unmemoized version if appropriate. It returns a reference to the unmemoized version of the function.

If you ask it to unmemoize a function that was never memoized, it croaks.

flush_cache

`flush_cache(function)` will flush out the caches, discarding *all* the cached data. The argument may be a function name or a reference to a function. For finer control over when data is discarded or expired, see the documentation for `Memoize::Expire`, included in this package.

Note that if the cache is a tied hash, `flush_cache` will attempt to invoke the `CLEAR` method on the hash. If there is no `CLEAR` method, this will cause a run-time error.

An alternative approach to cache flushing is to use the `HASH` option (see above) to request that `Memoize` use a particular hash variable as its cache. Then you can examine or modify the hash at any time in any way you desire. You may flush the cache by using `%hash = ()`.

CAVEATS

Memoization is not a cure-all:

- Do not memoize a function whose behavior depends on program state other than its own arguments, such as global variables, the time of day, or file input. These functions will not produce correct results when memoized. For a particularly easy example:

  ```
  sub f {
    time;
  }
  ```

  This function takes no arguments, and as far as `Memoize` is concerned, it always returns the same result. `Memoize` is wrong, of course, and the memoized version of this function will call `time` once to get the current time, and it will return that same time every time you call it after that.

- Do not memoize a function with side effects.

  ```
  sub f {
    my ($a, $b) = @_;
           my $s = $a + $b;
    print "$a + $b = $s.\n";
  }
  ```

This function accepts two arguments, adds them, and prints their sum. Its return value is the numuber of characters it printed, but you probably didn't care about that. But `Memoize` doesn't understand that. If you memoize this function, you will get the result you expect the first time you ask it to print the sum of 2 and 3, but subsequent calls will return 1 (the return value of `print`) without actually printing anything.

- Do not memoize a function that returns a data structure that is modified by its caller.

  Consider these functions: `getusers` returns a list of users somehow, and then `main` throws away the first user on the list and prints the rest:

```
sub main {
  my $userlist = getusers();
  shift @$userlist;
  foreach $u (@$userlist) {
    print "User $u\n";
  }
}

sub getusers {
  my @users;
  # Do something to get a list of users;
  \@users;  # Return reference to list.
}
```

  If you memoize `getusers` here, it will work right exactly once. The reference to the users list will be stored in the memo table. `main` will discard the first element from the referenced list. The next time you invoke `main`, `Memoize` will not call `getusers`; it will just return the same reference to the same list it got last time. But this time the list has already had its head removed; `main` will erroneously remove another element from it. The list will get shorter and shorter every time you call `main`.

  Similarly, this:

```
$u1 = getusers();
$u2 = getusers();
pop @$u1;
```

  will modify $u2 as well as $u1, because both variables are references to the same array. Had `getusers` not been memoized, $u1 and $u2 would have referred to different arrays.

- Do not memoize a very simple function.

  Recently someone mentioned to me that the Memoize module made his program run slower instead of faster. It turned out that he was memoizing the following function:

```
sub square {
  $_[0] * $_[0];
}
```

  I pointed out that `Memoize` uses a hash, and that looking up a number in the hash is necessarily going to take a lot longer than a single multiplication. There really is no way to speed up the `square` function.

  Memoization is not magical.

PERSISTENT CACHE SUPPORT

You can tie the cache tables to any sort of tied hash that you want to, as long as it supports `TIEHASH`, `FETCH`, `STORE`, and `EXISTS`. For example,

```
tie my %cache => 'GDBM_File', $filename, O_RDWR|O_CREAT, 0666;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

works just fine. For some storage methods, you need a little glue.

SDBM_File doesn't supply an EXISTS method, so included in this package is a glue module called Memoize::SDBM_File which does provide one. Use this instead of plain SDBM_File to store your cache table on disk in an SDBM_File database:

```
        tie my %cache => 'Memoize::SDBM_File', $filename, O_RDWR|O_CREAT,
0666;
        memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

NDBM_File has the same problem and the same solution. (Use Memoize::NDBM_File instead of plain NDBM_File.)

Storable isn't a tied hash class at all. You can use it to store a hash to disk and retrieve it again, but you can't modify the hash while it's on the disk. So if you want to store your cache table in a Storable database, use Memoize::Storable, which puts a hashlike front-end onto Storable. The hash table is actually kept in memory, and is loaded from your Storable file at the time you memoize the function, and stored back at the time you unmemoize the function (or when your program exits):

```
        tie my %cache => 'Memoize::Storable', $filename;
 memoize 'function', SCALAR_CACHE => [HASH => \%cache];

        tie my %cache => 'Memoize::Storable', $filename, 'nstore';
 memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

Include the `nstore' option to have the Storable database written in `network order'. (See *Storable* for more details about this.)

The flush_cache() function will raise a run-time error unless the tied package provides a CLEAR method.

EXPIRATION SUPPORT

See Memoize::Expire, which is a plug-in module that adds expiration functionality to Memoize. If you don't like the kinds of policies that Memoize::Expire implements, it is easy to write your own plug-in module to implement whatever policy you desire. Memoize comes with several examples. An expiration manager that implements a LRU policy is available on CPAN as Memoize::ExpireLRU.

BUGS

The test suite is much better, but always needs improvement.

There is some problem with the way goto &f works under threaded Perl, perhaps because of the lexical scoping of @_. This is a bug in Perl, and until it is resolved, memoized functions will see a slightly different caller() and will perform a little more slowly on threaded perls than unthreaded perls.

Some versions of DB_File won't let you store data under a key of length 0. That means that if you have a function f which you memoized and the cache is in a DB_File database, then the value of f() (f called with no arguments) will not be memoized. If this is a big problem, you can supply a normalizer function that prepends "x" to every key.

MAILING LIST

To join a very low-traffic mailing list for announcements about Memoize, send an empty note to mjd-perl-memoize-request@plover.com.

AUTHOR

Mark-Jason Dominus (`mjd-perl-memoize+@plover.com`), Plover Systems co.

See the `Memoize.pm` Page at http://www.plover.com/~mjd/perl/Memoize/ for news and upgrades. Near this page, at http://www.plover.com/~mjd/perl/MiniMemoize/ there is an article about memoization and about the internals of Memoize that appeared in The Perl Journal, issue #13. (This article is also included in the Memoize distribution as `article.html'.)

My upcoming book will discuss memoization (and many other fascinating topics) in tremendous detail. It will be published by Morgan Kaufmann in 2002, possibly under the title *Perl Advanced Techniques Handbook*. It will also be available on-line for free. For more information, visit http://perl.plover.com/book/ .

To join a mailing list for announcements about `Memoize`, send an empty message to `mjd-perl-memoize-request@plover.com`. This mailing list is for announcements only and has extremely low traffic---about two messages per year.

COPYRIGHT AND LICENSE

THANK YOU

Many thanks to Jonathan Roy for bug reports and suggestions, to Michael Schwern for other bug reports and patches, to Mike Cariaso for helping me to figure out the Right Thing to Do About Expiration, to Joshua Gerth, Joshua Chamas, Jonathan Roy (again), Mark D. Anderson, and Andrew Johnson for more suggestions about expiration, to Brent Powers for the Memoize::ExpireLRU module, to Ariel Scolnicov for delightful messages about the Fibonacci function, to Dion Almaer for thought-provoking suggestions about the default normalizer, to Walt Mankowski and Kurt Starsinic for much help investigating problems under threaded Perl, to Alex Dudkevich for reporting the bug in prototyped functions and for checking my patch, to Tony Bass for many helpful suggestions, to Jonathan Roy (again) for finding a use for `unmemoize()`, to Philippe Verdret for enlightening discussion of `Hook::PrePostCall`, to Nat Torkington for advice I ignored, to Chris Nandor for portability advice, to Randal Schwartz for suggesting the '`flush_cache` function, and to Jenda Krynicky for being a light in the world.

Special thanks to Jarkko Hietaniemi, the 5.8.0 pumpking, for including this module in the core and for his patient and helpful guidance during the integration process.