CHAPTER 3

CACHING AND MEMOIZATION

We saw in Section 1.8 that a natural recursive function can sometimes perform extremely badly. An easy and general solution to many of these performance problems, as well as some that arise in nonrecursive contexts, is *caching*.

Let's consider a program that converts images from one format to another. Specifically, let's imagine that the input is in the popular GIF format, and that the output is something we're going to send to the printer. The printer is not the little machine that sits on your desk; it's a big company with giant printing presses that will print one million copies of some magazine by Thursday afternoon.

The printer wants the images in a special CMYK format. CMYK stands for "Cyan-Magenta-Yellow-Black," which are the four colors of the special printer's inks that the printer uses to print the magazines.[1] However, the colors in the GIF image are specified as RGB values, which are the intensities of red, green, and blue light that will be emitted by our computer monitor when it displays the image. We need to convert the RGB values that are suitable for the monitor into CMYK values that are appropriate for printing.

The conversion is just a matter of simple arithmetic:

CODE LIBRARY
RGB-CMYK

```
sub RGB_to_CMYK {
  my ($r, $g, $b) = @_;
  my ($c, $m, $y) = (255-$r, 255-$g, 255-$b);
  my $k = $c < $m ? ($c < $y ? $c : $y)
                  : ($m < $y ? $m : $y);  # Minimum
  for ($c, $m, $y) { $_ -= $k }
```

1 "K" is for "black"; the printers don't use "B" because "B" is for "blue."

```
    [$c, $m, $y, $k];
}
```

Now we write the rest of the program, which opens the GIF file, reads the pixels one at a time, calls `RGB_to_CMYK()` for each pixel, and writes out the resulting CMYK values in the appropriate format.

There's a minor problem here. Let's suppose that the GIF image is 1024 pixels wide and 768 pixels high, for a total of 786,432 pixels. We will have made 786,432 calls to `RGB_to_CMYK()`. That seems all right, except for one thing: Because of the way the GIF format is defined, no GIF image ever contains more than 256 different colors. That means that at least 786,176 of our 786,432 calls were a waste of time, because we were doing the same computations that we had already done before. If we could figure out how to save the results of our `RGB_to_CMYK()` computations and recover them when appropriate, we might win back some performance.

In Perl, whenever we consider the problem of checking whether we've seen something already, the solution will almost always involve a hash. This is no exception. If we can use the RGB value as a hash key, we can make a hash that records whether we have seen a particular set of RGB values before, and if so, what the corresponding CMYK value was. Then our program logic will go something like this: To convert a set of RGB values to a set of CMYK values, first look up the RGB values in the hash. If they're not there, do the calculation as before, store the result in the hash, and return it as usual. If the values are in the hash, then just get the CMYK values from the hash and return them without doing the calculation a second time.

The code will look something like this:

CODE LIBRARY
RGB-CMYK-caching

```
my %cache;

sub RGB_to_CMYK {
  my ($r, $g, $b) = @_;
  my $key = join ',', $r, $g, $b;
  return $cache{$key} if exists $cache{$key};
  my ($c, $m, $y) = (255-$r, 255-$g, 255-$b);
  my $k = $c < $m ? ($c < $y ? $c : $y)
                  : ($m < $y ? $m : $y);  # Minimum
  for ($c, $m, $y) { $_ -= $k }
  return $cache{$key} = [$c, $m, $y, $k];
}
```

Suppose we call `RGB_to_CMYK()` with arguments `128,0,64`. The first time we do this, the function will look in the `%cache` hash under the key `'128,0,64'`;

there won't be anything there, so it will continue through the function, performing the calculation as usual, and, on the last line, store the result into `$cache{'128,0,64'}`, and return the result. The second time we call the function with the same arguments, it computes the same key, and returns the value of `$cache{'128,0,64'}` without doing any extra calculation. When we find the value we need in the cache without further calculation, that is called a *cache hit*; when we compute the right key but find that no value is yet cached under that key, it is called a *cache miss*.

Of course there's a possibility that the extra program logic and the hash lookups will eat up the gains that we got from avoiding the computation. Whether this is true depends on how time-consuming the original computation was and on the likelihood of cache hits. When the original computation takes a long time, caching is more likely to be a benefit. To be sure, we should run a careful benchmark of both versions of the function. But to help develop an intuition for the kinds of tradeoffs to expect, we will look briefly at the theory.

Suppose the typical call to the real function takes time f. The average time taken by the memoized version will depend on two additional parameters: K, the cache management overhead, and h, the probability of getting a cache hit on any particular call. In the extreme case where we never get a cache hit, h is zero; as the likelihood of cache hits increases, h approaches 1.

For a memoized function, the average time per call will be at least K, since every call must check the cache, plus an additional f if there is a cache miss, for a total of $K + (1 - h)f$. The unmemoized version of the function, of course, always takes time f, so the difference is simply $hf - K$. If $K < hf$, the memoized version of the function will be faster than the unmemoized version. To speed up the memoized version, we can increase the cache hit rate h, or decrease the cache management overhead K. When f is large, it is easier to achieve $K < hf$, so caching is more likely to be effective when the original function takes a long time to run. In the worst case, we never get any cache hits, and $h = 0$, so the "speedup" is actually a slowdown of $-K$.

3.1 CACHING FIXES RECURSION

We saw in Section 1.8 that recursive functions sometimes blow up and take much too long, even on simple inputs, and that the Fibonacci function is an example of this problem:

```
# Compute the number of pairs of rabbits alive in month n
sub fib {
```

```
    my ($month) = @_;
    if ($month < 2) { 1 }
    else {
        fib($month-1) + fib($month-2);
    }
}
```

As we saw in Section 1.8, this function runs slowly for most arguments, because it wastes time recomputing results it has already computed. For example, `fib(20)` needs to compute `fib(19)` and `fib(18)`, but `fib(19)` *also* computes `fib(18)`, as well as `fib(17)`, which is also computed once by each of the calls to `fib(18)`. This is a common problem with recursive functions, and it is fixed by caching. If we add caching to `fib`, then instead of recomputing `fib(18)` over again from scratch the second time it is needed, `fib` will simply retrieve the cached result of the first computation of `fib(18)`. It won't matter that we try to compute `fib(17)` three times or `fib(16)` five times because the work will be done only once, and the cached results will be retrieved quickly when they are needed again.

3.2 INLINE CACHING

The most straightforward way to add caching to a function is to give the function a private hash. In this example, we could use an array instead of a hash, since the argument to `fib()` is always a non-negative integer. But in general, we will need to use a hash, so that's what we'll see here:

CODE LIBRARY
fib-cached

```
# Compute the number of pairs of rabbits alive in month n
{ my %cache;
  sub fib {
    my ($month) = @_;
    unless (exists $cache{$month}) {
      if ($month < 2) { $cache{$month} = 1 }
      else {
        $cache{$month} = fib($month-1) + fib($month-2);
      }
    }
    return $cache{$month};
  }
}
```

Here `fib` gets the same argument as before. But instead of going into the recursive Fibonacci calculation immediately, it checks the cache first. The cache is a hash, `%cache`. When the function computes a Fibonacci number `fib($month)`, it will store the value in `$cache{$month}`. Later calls to `fib()` will check to see if there is a value in the cache hash. This is the purpose of the `exists $cache{$month}` test. If the cache element is absent, the function has never been called before for this particular value of `$month`. The code inside the `unless` block is just the ordinary Fibonacci computation, including recursive calls if necessary. However, once the function has computed the answer, it doesn't return it immediately; instead, it inserts the value into the cache hash in the appropriate place. For example, `$cache{$month} = 1` takes care of populating the cache when `$month < 2` is true.

At the end of the function, `return $cache{$month}` returns the cached value, whether the function just inserted it or it was there to begin with.

With these changes, the `fib` function is fast. The excessive recursion problem we saw in Chapter 1 simply goes away. The problem was caused by the repeated recomputation of results; adding caching behavior prevents any recomputation from occurring. When the function tries to recompute a result it has computed already, it immediately gets the value from the cache instead.

3.2.1 Static Variables

Why is `%cache` outside of `fib` instead of inside, and why is there a bare block around `%cache` and `fib`?

If `%cache` were declared inside of `fib`, like this:

```
sub fib {
  my %cache;
  ...
}
```

then the cache would not work, because a new, fresh `%cache` variable would be created on every call to `fib`, and thrown away when `fib` returned. By declaring `%cache` outside of any function, we tell Perl that we want only one instance of `%cache`, created when the program is first compiled and destroyed only when the program is finished. This allows `%cache` to accumulate values and retain them in between calls to `fib`. A variable like `%cache` that has been declared outside all the functions is called a *static variable* because its value stays the same unless it is explicitly changed, and also because a similar feature of the C language is activated with the keyword `static`.

`%cache` has been declared with `my`, so it is lexically scoped. By default, its scope will continue to the end of the file. If we had defined any functions after

`fib`, they would also be able to see and modify the cache. But this isn't what we want; we want the cache to be completely private to `fib`. Enclosing both `%cache` and `fib` in a separate block accomplishes this. The scope of `%cache` extends only to the end of the block, which contains only `fib` and nothing else.

3.3 GOOD IDEAS

There aren't too many ideas that are both good and simple. The few that we have are used everywhere. Caching is one of these. Your web browser caches the documents it retrieves from the network. When you ask for the same document a second time, the browser retrieves the cached copy from local disk or memory, which is fast, instead of downloading it again. Your domain name server caches the responses that it receives from remote servers. When you look up the same name a second time, the local server has the answer ready and doesn't have to carry on another possibly time-consuming network conversation. When your operating system reads data from the disks, it probably caches the data in memory, in case it's read again; when your CPU fetches data from memory, it caches the data in a special cache memory that is faster than the regular main memory.

Caching comes up over and over in real programs. Almost any program will contain functions where caching might yield a performance win. But the best property of caching is that it's *mechanical*. If you have a function, and you would like to speed it up, you might rewrite the function, or introduce a better data structure, or a more sophisticated algorithm. This might require ingenuity, which is always in short supply. But adding caching is a no-brainer; the caching transformation is always pretty much the same. This:

```
sub some_function {
  $result = some computation involving @_;
  return $result;
}
```

turns into this:

```
{ my %cache;
  sub some_function_with_caching {
    my $key = join ',', @_;
    return $cache{$key} if exists $cache{$key};
    $result = the same computation involving @_;
```

```
    return $cache{$key} = $result;
  }
}
```

The transformation is almost exactly the same for every function. The only part that needs to vary is the `join ',', @_` line. This line is intended to turn the function's argument array into a string, suitable for a hash key. Turning arbitrary values into strings like this is called *serialization* or *marshalling*.[2] The preceding `join ',', @_` example works only for functions whose arguments are numbers or strings that do not contain commas. We will look at the generation of cache keys in greater detail later on.

3.4 MEMOIZATION

Adding the caching code to functions is not very much trouble. And as we saw, the changes required are the same for almost any function. Why not, then, get the computer to do it for us? We would like to tell Perl that we want caching behavior enabled on a function. Perl should be able to perform the required transformation automatically. Such automatic transformation of a function to add caching behavior is called *memoization* and the function is said to be *memoized*.[3]

The standard `Memoize` module, which I wrote, does this. If the `Memoize` module is available, we do not need to rewrite the `fib` code at all. We simply add two lines at the top of our program:

CODE LIBRARY
fib-automemo

```
use Memoize;
memoize 'fib';
# Compute the number of pairs of rabbits alive in month n
sub fib {
  my ($month) = @_;
  if ($month < 2) { 1 }
  else {
      fib($month-1) + fib($month-2);
  }
}
```

2 Data marshalling is so named because it was first studied in 1962 by Edward Waite Marshall, then with the General Electric corporation.

3 The term *memoization* was coined in 1968 by Donald Michie.

`fib` now exhibits the caching behavior. The code is exactly the same as our original slow version, but the function is no longer slow.

3.5 THE MEMOIZE MODULE

This book isn't about the internals of Perl modules, but some of the techniques used internally by `Memoize` are directly relevant to things we'll be doing later on, so we'll have a short excursion now.

`Memoize` gets a function name (or reference) as its argument. It manufactures a new function that maintains a cache and looks up its arguments in the cache. If the new function finds the arguments in the cache, it returns the cached value; if not, it calls the original function, saves the return value in the cache, and returns it to the original caller.

Having manufactured this new function, `Memoize` then installs it into the Perl symbol table in place of the original function so that when you think you're calling the original function, you actually get the new cache manager function instead.

Rather than looking into the innards of the real `Memoize` module, which is a 350-line monster, we'll see a tiny, stripped-down memoizer. The most important thing we'll get rid of is the part of the code that deals with the Perl symbol table. (We'll do this manually.) Instead, we'll have a `memoize` function whose argument is a reference to the subroutine we want to memoize, and which returns a reference to the memoized version — that is, to the cache manager function:

CODE LIBRARY
memoize

```
sub memoize {
  my ($func) = @_;
  my %cache;
  my $stub = sub {
    my $key = join ',', @_;
    $cache{$key} = $func->(@_) unless exists $cache{$key};
    return $cache{$key};
  };
  return $stub;
}
```

To call this, we first use:

```
$fastfib = memoize(\&fib);
```

Now `$fastfib` is the memoized version of `fib()`. To install the memoized version of `fib()` in the symbol table in place of the original, we would write `*fib = memoize(\&fib)`. In this example, the installation is necessary if we want to calculate Fibonacci numbers quickly. Just creating a memoized version of `fib()` isn't enough, because the recursive calls inside of `fib()` are calling the function named `fib()`, and until we do the `*fib` assignment, this is still the old, slow, unmemoized version.

How does `memoize` work? We pass it a reference to `fib`, and `memoize` sets up a private `%cache` variable to hold cached data. Then it manufactures a *stub function*, temporarily stored in `$stub`, which it returns to its caller. This stub function is actually the memoized version of `fib`; the caller of `memoize` gets back a reference to it, which we stored in `$fastfib` in the preceding example.

When we invoke `$fastfib`, we actually get the stub function that was previously manufactured by `memoize`. The stub function assembles a hash key by joining the function arguments together with commas; then it looks in the cache to see if the key is a familiar one. If so, the stub returns the cached value immediately.

If the hash key isn't found in the hash, the stub function invokes the original function via `$func->(@_)`, gets the result, stores it in the cache, and returns it (see Figure 3.1).

3.5.1 Scope and Duration

There are some subtleties here. First, suppose we call `memoize(\&fib)` and get back `$fastfib`. Then we call `$fastfib`, which makes use of `$func`. A common question is why `$func` didn't go out of scope when `memoize` returned.

This question betrays a common misconception about scope. A variable has two parts: a name and a value.[4] When you associate a name with a value, you get a variable. Such an association is called a *binding*; we also say that the name is *bound* to its value.

There are two things that might go wrong with our attempt to use `$func` after `memoize` returns: The value might have been destroyed, or the binding might have changed, so that the name refers to the wrong value, or to nothing at all.

4 This is not precisely accurate. In imperative languages like Perl, a variable is an association between a name and the part of the computer's memory *in which the value will be stored*. For purposes of our discussion, this distinction is unimportant.

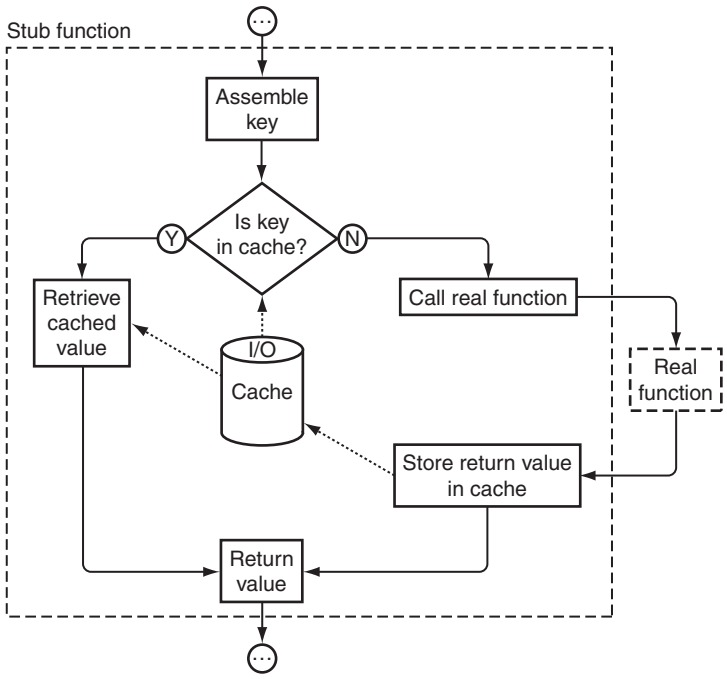


FIGURE 3.1 Calling a memoized function.

SCOPE

scope is the part of the program's source code in which a certain binding is in force. Inside the scope of a binding, the name and value are associated; outside this scope, the binding is *out of scope* and the name and value are no longer associated. The name might mean something else, or nothing at all.

When `memoize` is entered, the `my $func` declaration creates a new, fresh scalar value and binds the name `$func` to it. The scope of the name declared with `my`, such as `$func`, begins on the statement following the `my` declaration, and ends at the end of the smallest enclosing block. In this case, the smallest enclosing block is the one labeled `sub memoize`. Inside this block, `$func` refers to the lexical variable just created; outside, it refers to something else, probably the unrelated global variable `$func`. Since the stub that uses `$func` is inside this block, there's no scope problem; `$func` is in scope inside of the stub, and the name `$func` retains its binding.

Outside the `sub memoize` block, `$func` means something different, but the stub is inside the block, not outside. Scope is *lexical*, which means that

it's a property of the static program text, not a property of the order in which things execute. The fact that the stub is *called* from outside the `sub memoize` block is irrelevant; its code is "physically within" the scope of the `$func` binding.

The situation is the same for `%cache`.

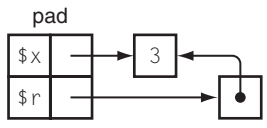
DURATION

Most people who ask whether `$func` is out of scope are worried about a different problem, which is not a scope issue, but instead concerns something quite different, called *duration*. The duration of a value is the period of time during the program's execution in which the value is valid for use. In Perl, when a value's duration is up, it is destroyed, and the garbage collector makes its memory available for re-use.

The important thing to know about duration is that it is almost completely unrelated to issues of names. In Perl, a value's duration lasts until there are no outstanding references to it. If the value is stored in a named variable, that counts as a reference, but there are other kinds of references. For example:

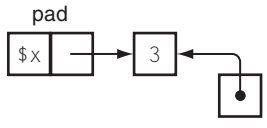
```
my $x;
{
  $x = 3;
  my $r = \$x;
}
```

Here there is a scalar with the value 3. At the end of the block, there are two references to it:

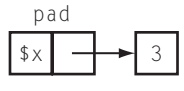


The *pad* is the data structure that Perl uses internally to represent bindings of `my` variables. (A different structure is used for global variables.) One reference to the 3 is from the pad itself, because the name `$x` is bound to the value 3. The other reference to the 3 is from the reference value that is bound to `$r`.

When control leaves the block, `$r` goes out of scope, so the binding of `$r` to its value is dissolved. Internally, Perl deletes the `$r` binding from the pad:

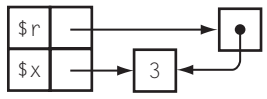


Now there's no reference to the reference value that used to be stored in `$r`. Since nothing refers to it anymore, its duration is over and Perl destroys it immediately:

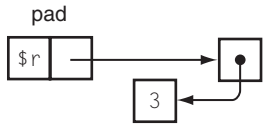


This is typical: A variable's name goes out of scope, and its value is destroyed immediately after. Much of the confusion between scope and duration is probably caused by the ubiquity of this simple example. But scope and duration are not always so closely joined, as the next example will show:

```
my $r;
{
  my $x = 3;
  $r = \$x;
}
```



When control leaves the block, the binding of `$x` is dissolved and `$x` is deleted from the pad:



Unlike in the previous example, the unbound value persists indefinitely because it is still referred to by the reference bound to `$r`. Its duration will not be over until this reference is gone.

This separation of scope and duration is an essential property of Perl variables. For example, a common pattern in Perl object-oriented constructor functions is:

```
sub new {
  ...
  my %self;
  ...
  return \%self;
}
```

This constructor manufactures a hash, which is the object itself; then it returns a reference to the hash. Even though the name `%self` has gone out of scope, the object persists as long as the caller holds a reference to it. The analogous code in C is erroneous, because in C, the duration of an auto variable ends with its scope:

```
/* This is C */
struct st_object *new(...) {
  struct st_object self;
  ...
  return &self; /* expect a core dump */
}
```

Now let's return to `memoize`. When `memoize` returns, `$func` does indeed go out of scope. But the value is not destroyed, because there is still an outstanding reference from the stub. To really understand what is going on, we need to take a peek into Perl's internals (see Figure 3.2).

The stub is represented by the double box at the top center of the diagram. In Perl jargon, this box is called a *CV*, for "code value"; it is the internal representation of a coderef. (The coderef bound to `$func` is visible on the right-hand side of the diagram.) A CV is essentially a pair of pointers: one points to the code for the subroutine, and the other points to the pad that was active at the moment that the subroutine was defined. The binding of `$func` won't be destroyed until the pad it's in is destroyed. The pad won't be destroyed because there is a reference to it from the CV. The CV won't be destroyed because the caller stored it in the caller's pad by assigning it to `$fastfib`.

Perl knows that the stub might someday be invoked, and if it is, it might examine the value of `$func`. As long as the stub exists, the value of `$func` must be

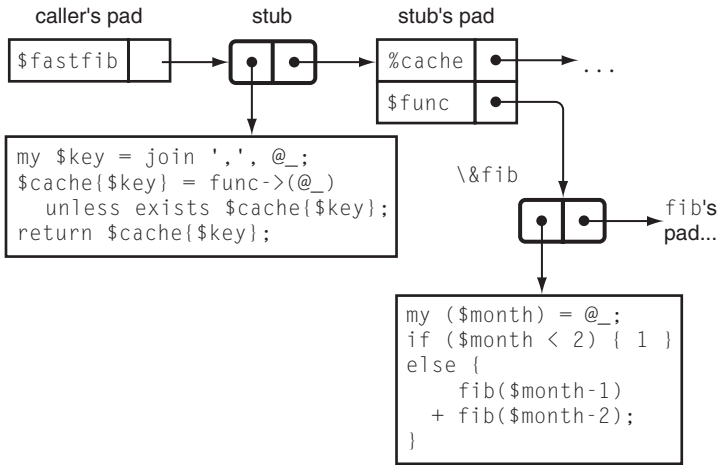


FIGURE 3.2 The data structure manufactured by `memoize`.

preserved intact. There is a reference to the stub stored in `$fastfib`, and as long as the reference is there, the stub must be preserved. Similarly, the cache `%cache` persists as long as the stub does.

3.5.2 Lexical Closure

Now another point might be worrying you. Since the value of `$func` persists as long as the stub does, what happens if we call `memoize` a second time, while the first stub is still extant? Will the assignment to `$func` in the second call clobber the value that the first stub was using?

The answer is no; everything works perfectly. This is because Perl's anonymous functions have a property called *lexical closure*. When an anonymous function is created, Perl packages up its pad, including all the bindings that are in scope, and attaches them to the CV. A function packaged up with an environment in this way is called a *closure*.

When the stub is invoked, the original environment is temporarily reinstated, and the stub function code is run in the environment that was in force at the time the stub was defined. Lexical closure means that an anonymous function carries its native environment wherever it goes, just like some tourists I have met.

The first time we call `memoize`, to memoize `fib()`, a new pad is set up for the bindings of `%cache` and `$func`, new storage is allocated for these new variables, and `$func` is initialized. Then the stub is created; the pad is attached to the CV for the stub, and the CV (let's call it `fastfib()`) is returned back to the caller.

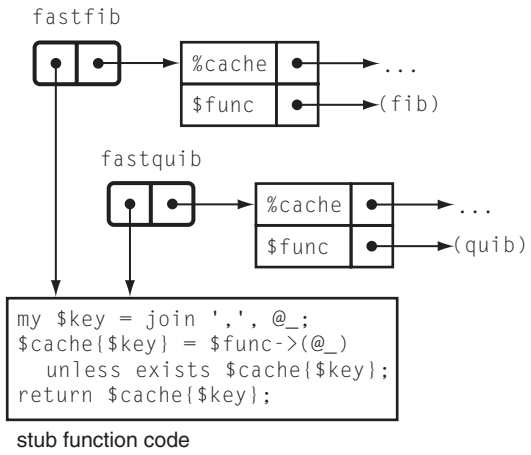


FIGURE 3.3 After two calls to `memoize`.

Now let's call `memoize` a second time, this time to memoize `quib()` instead of `fib()` (see Figure 3.3). Once again, a new pad is created and fresh `%cache` and `$func` variables are bound into it. A CV is created (which we'll call `fastquib()`) that contains a pointer to the new pad. The new pad is completely unrelated to the pad that was attached to `fastfib()`.

When we invoke `fastfib`, `fastfib`'s pad is temporarily reinstated, and `fastfib`'s code is executed. The code makes use of variables named `%cache` and `$func`, and these are looked up in `fastfib`'s pad. Perhaps some data is stored into `%cache` at this time. Eventually, `fastfib` returns, and the old pad comes back into force.

Then we invoke `fastquib`, and almost the same thing happens. `fastquib`'s pad is reinstated, with its own notion of `%cache` and `$func`. `fastquib`'s code is run, and it too makes use of variables named `%cache` and `$func`. These are looked up in `fastquib`'s pad, which has no connection to `fastfib`'s pad. Data stored into `fastfib`'s `%cache` is completely inaccessible to `fastquib`.

Because the code part of the CV is read-only, it is shared between several CVs. This saves memory. When a CV's duration is over, its pad is garbage-collected.

Figure 3.4 shows a simpler example.

CODE LIBRARY
closure-example

```
sub make_counter {
  my $n = shift;
  return sub { print "n is ", $n++ };
}

my $x = make_counter(7);
my $y = make_counter(20);
```

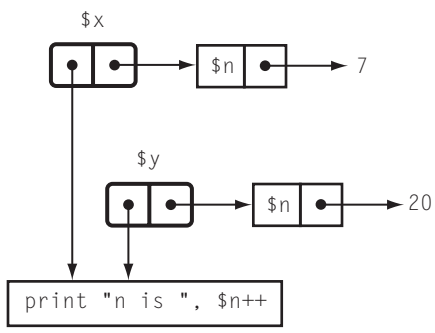


FIGURE 3.4 After two calls to `make_counter`.

```
$x->(); $x->(); $x->();
$y->(); $y->(); $y->();
$x->();
```

`$x` now contains a closure whose code is `print "n is ", $n++` and whose environment contains a variable `$n`, set to 7. If we invoke `$x` a few times:

```
$x->(); $x->(); $x->();
```

the result is

```
n is 7
n is 8
n is 9
```

The new picture is shown in Figure 3.5.

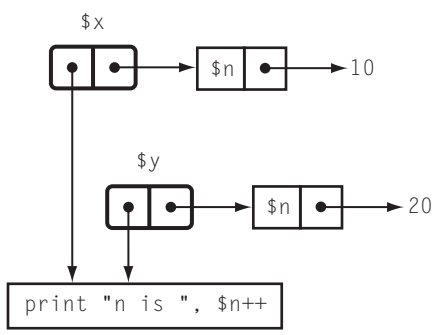


FIGURE 3.5

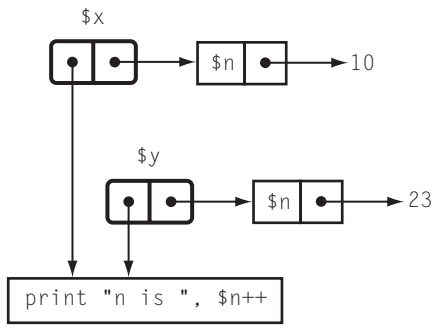


FIGURE 3.6

Now let's run `$y` a few times:

```
$y->(); $y->(); $y->();
```

The same code runs, but this time the name `$n` is looked up in `$y`'s pad instead of in `$x`'s pad:

```
n is 20
n is 21
n is 22
```

The new picture is shown inFigure 3.6.

Now let's run `$x` again:

```
n is 10
```

The `$n` here is the same variable as it was the first three times we invoked `$x`, and it has retained its value.

3.5.3 Memoization Again

All of the foregoing discussion was by way of explaining just why our `memoize` function worked. While it's tempting to dismiss this as a triviality — "Of course it worked!" — it's worth noticing that in many languages it won't work and can't be made to work. Several important and complex features had to operate together: delayed garbage collection, bindings, generation of anonymous subroutines, and lexical closure. If you tried to implement a function like `memoize` in C, for

example, you would get stuck, because C doesn't have any of those features. (See Section 3.11.)

3.6 CAVEATS

(That's Latin for "warnings.")

Clearly, memoization is not a suitable solution for all performance problems. It is not even applicable to all functions. There are several kinds of functions that should not be memoized.

3.6.1 Functions Whose Return Values Do Not Depend on Their Arguments

Memoization is most suitable for functions whose return values depend only on their arguments. Imagine the foolishness of memoizing a time-of-day function: The first time you called it, you would get the time of day; subsequent calls would return the *same* time. Similarly, imagine the perversity of a memoized random number generator.

Or imagine a function whose return value indicates a success or failure of some sort. You do not want such a function to be memoized and return the same value every time it is called.

However, memoization is suitable for some such functions. For example, it might be useful to memoize a function whose result depends on the current hour of the day, if the program will run for a long time. (See Section 3.7 for details about how to handle this.)

3.6.2 Functions with Side Effects

Many functions are called not for their return values but for their side effects. Suppose you have written a program that formats a computer uptime report and delivers the report to the printer to be printed. Probably the return value is not interesting, and caching it is silly. Even if the return value is interesting, memoization is still inappropriate. The function might complete much more quickly after the first run, because of the memoization, but your boss would not be impressed, because it would have returned the old cached return value immediately, without bothering to actually print the report.[5]

5 I sometimes enjoy the mind-bending exercise of imagining the result of memoizing the Unix `fork()` function.

3.6.3 Functions That Return References

This problem is a little more subtle. Functions that return references to values that may be modified by their callers must not be memoized.

To see the potential problem, consider this example:

```
use Memoize;

sub iota {
  my $n = shift;
  return [1 .. $n];
}
memoize 'iota';

$i10 = iota(10);
$j10 = iota(10);
pop @$i10;
print @$j10;
```

The first call to `iota(10)` generates a new, fresh anonymous array of the numbers from 1 to 10, and returns a reference to this array. This reference is automatically placed in the cache, and is also stored into `$i10`. The second call to `iota(10)` fetches the same reference from the cache and stores it into `$j10`. Both `$i10` and `$j10` now refer to the same array — we say that they are *aliases* for the array.

When we change the value of the array via the `$i10` alias, the change affects the value that is stored in `$j10`! This was probably not what the caller was expecting, and it would not have happened if we had not memoized `iota`. Memoization is supposed to be an optimization. This means it is supposed to speed up the program without changing its behavior.

The prohibition on memoizing functions that return references to values that may be modified by the caller probably applies most commonly to object-oriented constructor methods. Consider:

```
package Octopus;
sub new {
  my ($class, %args) = @_;
  $args{tentacles} = 8;
  bless \%args => $class;
}

sub name {
```

```
  my $self = shift;
  if (@_) { $self->{name} = shift }
  $self->{name};
}

my $junko = Octopus->new(favorite_food => "crab cakes");
$junko->name("Junko");
my $fenchurch = Octopus->new(favorite_food => "crab cakes");
$fenchurch->name("Fenchurch");

# This prints "Fenchurch" -- oops!
print "The name of the FIRST octopus is ", $junko->name, "\n";
```

Here the programmer is expecting to manufacture two different octopuses, one named "Junko" and the other "Fenchurch." Both octopuses enjoy crab cakes. Unfortunately, someone has foolishly decided to memoize `new()`, and since the arguments to it are the same on the second call, the memoization stub returns the cached return value from the first call, which is a reference to the "Junko" object. The programmer thinks that there are two octopuses, but really there is only one, masquerading as two.

Functions whose return values depend only on their arguments, and which do not have side effects, and which never return references are called *pure functions*. Caching techniques are most suitable for use with pure functions, although they can sometimes be used even with impure functions.

3.6.4 A Memoized Clock?

A simple and instructive example of a cached impure function is provided by Perl's `$^T` variable. Perl provides several convenient operators for files, such as `-M $filename`, which returns the amount of time, in days, since the file named by its argument was last modified. To compute this, Perl asks the operating system for the last-modification time, subtracts this from the current time, and converts to days. Since `-M` may be performed very frequently, it is important that it be fast: Consider:

```
@result = sort { -M $a <=> -M $b } @files;
```

which sorts a list of files by their last modification time. It's already expensive to look up the last-modified times for many files, and there's no need to make thousands of calls to the `time()` function on top of that cost. Even worse, the OS

may track the time to an accuracy of only one second, and if the system clock happens to advance during the execution of the `sort()`, the result list might be in the wrong order!

To avoid these problems, Perl does not look up the current time whenever a `-M` operation is performed. Instead, when the program is first run, Perl caches the current time in the special `$^T` variable, and uses that as the current time whenever `-M` is invoked. Most programs are short-lived, and most don't need exact accuracy in the results from `-M`, so this is usually a good idea. Certain long-running programs need to periodically refresh `$^T` by doing `$^T = time()`, to prevent the `-M` results from getting too far out of date. When caching an impure function, it is usually a good idea to provide an expiration regime, in which old cached values are eventually discarded and refreshed. It is also prudent to allow the programmer a way to flush the entire cache. The `Memoize` module provides the opportunity to plug in a cache expiry manager.

3.6.5 Very Fast Functions

I once talked to a programmer who complained that when he memoized his function, it got slower instead of faster. It turned out that the function he was trying to speed up was:

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

Caching, like all techniques, is a tradeoff. The potential benefit is that you make fewer calls to the original function. The cost is that your program must examine the cache on every call. Earlier, we saw the formula $hf - K$, which expresses the amount of time saved by memoization. If $hf < K$, then the memoized version will be slower than the unmemoized version. h is the cache hit rate and is between 0 and 1. f is the original running time of the function, and K is the average time needed to check the cache. If f is smaller than K, $hf < K$ will be inevitable. If the cost of examining the cache is larger than the cost of calling the original function, memoization does not make sense. You can't save time by eliminating "unnecessary" calls because it takes longer to find out that the call is unnecessary than it does to make the call in the first place.

In our `square` example, the function is doing a single multiplication. Checking a cache requires a hash lookup; this includes computation of a hash value (many multiplications and additions), then indexing into the hash bucket array, and possibly a search of a linked list. There is no way this is going to beat a single multiplication. In fact, almost nothing beats a single multiplication. You can't

speed up the `square` function, by memoization or any other technique, because it is already almost as fast as any function can possibly be.

3.7 KEY GENERATION

The memoizer we saw earlier has at least one serious problem. It needs to turn the function arguments into a hash key, and the way it does that is with `join`:

```
my $key = join ',', @_;
```

This works for functions with only one argument, and it works for functions whose arguments never contain commas, including all functions whose arguments are numbers. But if the function arguments may contain commas, it might fail, because the same key is computed for these two calls:

```
func("x,", "y");
func("x", ",y");
```

When the first call is made, the return value will be stored in the cache under the key `"x,,y"`. When the second call is made, the true function will not be consulted. Instead the cached value from the first call will be returned. But the function might have wanted to return a different value — the memoization code has confused these two argument lists, resulting in a false cache hit.

Since this can fail only for functions whose arguments may contain commas, it may not be a consideration. Even if the function arguments may contain commas, it's possible that there is some other character that they will never contain. The Perl special variable `$;` is sometimes used here. It normally contains character `#28`, which is the control–backslash character. If the key generator uses `join $; , @_`, it will fail only when the function arguments contain control–backslash; it is often possible to be sure this will never occur. But often we have a function whose argument could contain absolutely anything, and one of these partial hacks won't work reliably.

This can be fixed, because there's always a way to turn any data structure, such as an argument list, into a string in a faithful way, so that different structures become different strings.[6]

---

6 To see this, just realize that there must be some difference in the way the two structures are represented in memory, and that the computer's memory is itself nothing more than a very long string.

One strategy would be to use the `Storable` or `FreezeThaw` module to turn the argument list into a string. A much more efficient strategy is to use escape sequences:

```
my @args = @_;
s/([\\,])/\\$1/g for @args;
my $key = join ",", @args;
```

Here we insert a backslash character before every comma or backslash in the original arguments, then join the results together with unbackslashed commas. The problem calls we saw earlier are no longer problems, because the two argument lists are transformed to different keys: one to `'x\,,y'` and the other to `'x,\,y'`. (An exercise: Why is it necessary to put a backslash before every backslash character as well as before every comma?)

However, correctness has been bought at a stiff performance price. The escape character code is much slower than the simple `join` — about ten times slower even for a simple argument list such as `(1,2)` — and it must be performed on *every* call to the function. Normally, we laugh at people who are willing to trade correctness for speed, since it doesn't matter how quickly one is able to find the wrong answer. But this is an unusual circumstance. Since the only purpose of memoization is to speed up a function, we want the overhead to be as small as possible.

We'll adopt a compromise. The default behavior of `memoize` will be fast, but not correct in all cases. We'll give the user of `memoize` an escape hatch to fix this. If the user doesn't like the default key-generation method, they may supply an alternative, which `memoize` will use instead.

The change is simple:

CODE LIBRARY
memoize-norm1

```
sub memoize {
  my ($func, $keygen) = @_;
  my %cache;
  my $stub = sub {
    my $key = $keygen ? $keygen->(@_) : join ',', @_;
    $cache{$key} = $func->(@_) unless exists $cache{$key};
    return $cache{$key};
  };
  return $stub;
}
```

The stub returned by `memoize` looks to see if a `$keygen` function was supplied when the original function was memoized. If so, it uses the `keygen` function to

construct the hash key; if not, it uses the default method. The extra test is fairly cheap, but we can eliminate it if we want to by performing the test for `$keygen` once, at the time the function is memoized, instead of once for each call to the memoized function:

CODE LIBRARY
memoize-norm2

```
sub memoize {
  my ($func, $keygen) = @_;
  my %cache;
  my $stub = $keygen ?
    sub { my $key = $keygen->(@_);
          $cache{$key} = $func->(@_) unless exists $cache{$key};
          return $cache{$key};
        }
  :
    sub { my $key = join ',', @_;
          $cache{$key} = $func->(@_) unless exists $cache{$key};
          return $cache{$key};
        }
  ;
  return $stub;
}
```

We can pull an even better trick here. In these versions of `memoize`, `$keygen` is an anonymous function that has to be invoked on each call to the memoized function. Perl, unfortunately, has a relatively high overhead for function calls, and since the purpose of `memoize` is to speed things up, we'd like to avoid this if we can.

Perl's `eval` feature comes to the rescue here. Instead of specifying `$keygen` as a reference to a key-generation function, we'll pass in a string that contains the code to generate the key, and incorporate this code directly into the stub, rather than as a sub-function that the stub must call.

To use this version of `memoize`, we will say something like this:

```
$memoized = memoize(\&fib, q{my @args = @_;
                             s/([\\,])/\\$1/g for @args;
                             join ',', @args;
                            });
```

`memoize` will interpolate this bit of code into the appropriate place in a template of the memoized function (this is called *inlining*) and use `eval` to compile the

result into a real function:

CODE LIBRARY
memoize-norm3

```
sub memoize {
  my ($func, $keygen) = @_;
  $keygen ||= q{join ',', @_};

  my %cache;
  my $newcode = q{
    sub { my $key = do { KEYGEN };
          $cache{$key} = $func->(@_) unless exists $cache{$key};
          return $cache{$key};
        }
  };
  $newcode =~ s/KEYGEN/$keygen/g;
  return eval $newcode;
}
```

Here we used Perl's `q{...}` operator, which is identical with `'...'`, except that single-quote characters aren't special inside of `q{...}`. Instead, the `q{...}` construction ends at the first matching `}` character. If we hadn't used `q{...}` here, the third line would have been rather cryptic:

```
$keygen ||= 'join \',\', @_';
```

We used the `s///` operator to inline the value of `$keygen`, instead of simply interpolating it into a double-quoted string. This is slightly less efficient, but it needs to be done only once per memoized function, so it probably doesn't matter. The benefit is that with the `s///` technique, the `$newcode` variable is easy to read; if we had used string interpolation, it would have been:

```
my $newcode = "
  sub { my \$key = do { $keygen };
        \$cache{\$key} = \$func->(\@_) unless exists \$cache{\$key};
        return \$cache{\$key};
      }
";
```

Here the backslashes clutter up the code. A maintenance programmer reading this might not notice that `$keygen` is being interpolated even though everything else is backslashed. With the `s///` technique, `KEYGEN` stands out clearly.

For this example, the cache management overhead is about 37% lower with the inlining version of `memoize`.

It's easy to tweak this version so that it still accepts a function reference the way the previous one did:

CODE LIBRARY
memoize-norm4

```
sub memoize {
  my ($func, $keygen) = @_;
  my $keyfunc;
  if ($keygen eq '') {
    $keygen = q{join ',', @_}
  } elsif (UNIVERSAL::isa($keygen, 'CODE')) {
    $keyfunc = $keygen;
    $keygen = q{$keyfunc->(@_)};
  }
  my %cache;
  my $newcode = q{
    sub { my $key = do { KEYGEN };
          $cache{$key} = $func->(@_) unless exists $cache{$key};
          return $cache{$key};
        }
  };
  $newcode =~ s/KEYGEN/$keygen/g;
  return eval $newcode;
}
```

Here, if no key generator is supplied, we inline `join ',', @_` as usual. If `$keygen` is a function reference, we can't simply inline it, because it will turn into something useless like `CODE(0x436c1d)`. Instead, we save the function reference in the `$keyfunc` variable and inline some code that will call the function via `$keyfunc`.

The `UNIVERSAL::isa($keygen, 'CODE')` line requires some explanation. We want to test to see if `$keygen` is a code reference. The obvious way of doing that is:

```
if (ref($keygen) eq 'CODE') { ... }
```

Unfortunately, the Perl `ref` function is broken, because it confuses two different properties of its argument. If `$keygen` is a *blessed* code reference, the test above will fail, because `ref` will return the name of the class into which `$keygen` has been blessed. Using `UNIVERSAL::isa` avoids this problem. It's also possible, although much less likely, that the test could yield true for a non-code reference; this will happen if someone has been silly enough to bless the non-code reference into the class `CODE`.

3.7.1 More Applications of User-Supplied Key Generators

With any of these key-generation features, users of our `memoize` function have escape hatches if the `join` method doesn't work correctly for the function they are trying to memoize. They can substitute a key generator based on `Storable` or the escape-character method or whatever is appropriate.

User-supplied key generators solve the problems that may occur when two different argument lists hash to the same key. They also solve the converse problem, which occurs when two equivalent argument lists hash to different keys.

Consider a function whose argument is a hash, which might contain any, all, or none of the keys `A`, `B`, and `C`, each with an associated numeric value. Further, suppose that `B`, if omitted, defaults to 17, and `A` defaults to 32:

```
sub example {
  my %args = @_;
  $args{A} = 32 unless defined $args{A};
  $args{B} = 17 unless defined $args{B};
  # ...
}
```

Then the following calls are all equivalent:

```
example(C => 99);
example(C => 99, A => 32);
example(A => 32, C => 99);
example(B => 17, C => 99);
example(C => 99, B => 17);
example(A => 32, C => 99, B => 17);
example(B => 17, A => 32, C => 99);
(etc.)
```

The `join` method of key construction generates a different key for each of these calls (`"C,99"` versus `"A,32,C,99"` versus `"C,99,A,32"` and so forth). The cache manager will therefore miss opportunities to avoid calling the real `example()` function. A call to `example(A => 32, C => 99)` must produce the same result as a call to `example(C => 99, A => 32)`, but the cache manager doesn't know that, because the argument lists are superficially different. If we can arrange that equivalent argument lists are transformed to the same hash key, the cache manager will return the same value for `example(C => 99, A => 32)` that it had previously computed for `example(A => 32, C => 99)`, without the redundant

call to `example`. This will increase the cache hit rate h in the formula $hf - K$ that expresses the speed-up from memoization. The following key generator does the trick:

```
sub {
  my %h = @_;
  $h{A} = 32 unless defined $h{A};
  $h{B} = 17 unless defined $h{B};
  join ",", @h{'A','B','C'};
}
```

Each of the eight equivalent calls (of which `example(C => 99, A => 32)` was one) receives a key of `"32,17,99"` from this function. Here we pay an up-front cost: This key generator takes about ten times as long as the simple `join` generator, so the K in the $hf - K$ formula is larger. Whether this cost is repaid depends on how expensive it is to call the real function, f, and on the size of the increase of cache hits frequency, h. As usual, there is no substitute for benchmarking.

3.7.2 Inlined Cache Manager with Argument Normalizer

Here's an interesting trick we can play with the inlined key-generation code. Consider the following function, a variation on the example we just saw:

```
sub example {
  my ($a, $b, $c) = @_;
  $a = 32 unless defined $a;
  $b = 17 unless defined $b;
  # more calculation here ...
}
```

A suitable key generator might be:

```
  my ($a, $b, $c) = @_;
  $a = 32 unless defined $a;
  $b = 17 unless defined $b;
  join ',', $a, $b, $c;
```

It's a little irritating to have to repeat the code that sets the defaults for the arguments, and equally irritating to have to run it twice. If we change the

key-generation code as follows, we will be able to remove the argument checking from the example function:

```
$_[0] = 32 unless defined $_[0];
$_[1] = 17 unless defined $_[1];
join ',', @_;
```

When this is inlined into the `memoize` function, the result is:

```
sub { my $key = do { $_[0] = 32 unless defined $_[0];
                     $_[1] = 17 unless defined $_[1];
                     join ',', @_;
                   };
      $cache{$key} = $func->(@_) unless exists $cache{$key};
      return $cache{$key};
    }
```

Notice what happens here. The key is generated as before, but there is a side effect: `@_` is modified. If there is a cache miss, the memoized function calls `$func` with the modified `@_`. Since `@_` has been modified to include the default values already, we can omit the default-setting code from the original function:

```
sub example {
  my ($a, $b, $c) = @_;
## defaults set by key generation code
##  $a = 32 unless defined $a;
##  $b = 17 unless defined $b;
  # more calculation here ...
}
```

Of course, once we've modified the `example` function in this way, we can't turn the memoization off, because essential functionality has been moved into the key generator.

Another danger with this technique is that modifying `@_` can have peculiar effects back in the *calling* function. Elements of `@_` are aliased to the corresponding arguments back in the caller, and assigning to elements of `@_` in the memoized function can modify variables outside the memoized function. Here is a simple example:

```
sub set_to_57 {
  $_[0] = 57;
}
```

```
my $x = 119;
set_to_57($x);
```

This does set `$x` to 57, as if we had done `$x = 57`, even though the assignment is performed outside the scope of `$x` and shouldn't be able to affect it. Our assignments inside the key-generator code may have similar effects.

Sometimes this feature of Perl is useful, but most often it is more trouble than it is worth, and we normally avoid it. We do this by never operating on `@_` directly, but by copying its contents into a series of lexical variables as soon as the function is called:

```
sub safe_function {
  my ($n) = @_;
  $n = 57; # does *not* set $x to 57
}

my $x = 119;
safe_function($x);
```

By combining these techniques, we can get a version of the key-generation code that obviates the need for the default-setting code in the real function, but which is still safe:

```
memoize(\&example, q{
  my ($a, $b, $c) = @_;
  $a = 32 unless defined $a;
  $b = 17 unless defined $b;
  @_ = ($a, $b, $c);            # line 5
  join ',', @_;
});
```

The elements of `@_` are aliases for the arguments back in the calling function, but `@_` itself isn't. The assignment to `@_` on line 5 doesn't overwrite the values back in the caller; it discards the aliases entirely and replaces the contents of `@_` with the new values. This trick works only when the key-generation code is inlined into the memoized function; if the key-generation code is called as a subroutine, the change to `@_` has no effect after the subroutine returns.

3.7.3 Functions with Reference Arguments

Here's another problem solved by the custom key-generator feature. Consider this function:

```
sub is_in {
  my ($needle, $haystack) = @_;
  for my $item (@$haystack) {
    return 1 if $item == $needle;
  }
  return;
}
```

The function takes `$needle`, which is a number, and `$haystack`, which is a list of numbers, and returns true if and only if `$haystack` contains `$needle`. A typical call is:

```
if (is_in($my_id, \@employee_ids)) { ... }
```

We might like to try memoizing `is_in`, but a possible problem is that the `$haystack` argument is a reference. When it is handled by the `join` function, it turns into a string like `ARRAY(0x436c1d)`. If we later call `is_in()` with a `$haystack` argument that refers to a different array with the same contents, the hash key will be different, which may not be what we want; conversely, if the contents of `@employee_ids` change, the hash key will still be the same, which certainly isn't what we want. The key generator is generating the key from the identity of the array, but the `is_in()` function doesn't care about the identity of the array; it cares only about the contents. A more appropriate key-generation function in this case is:

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

Again, whether this actually produces a performance win depends on many circumstances that will be hard to foresee. When performance is important, it is essential to gather real data. Long experience has shown that even experts are likely to guess wrong about what is fast and what is slow.

3.7.4 Partitioning

The `find_share` function of Chapter 1 provides a convenient example of a function for which memoization fixes slow recursion, as well as requiring a custom

key generator:

```
sub find_share {
  my ($target, $treasures) = @_;
  return [] if $target == 0;
  return    if $target < 0 || @$treasures == 0;
  my ($first, @rest) = @$treasures;
  my $solution = find_share($target-$first, \@rest);
  return [$first, @$solution] if $solution;
  return         find_share($target       , \@rest);
}
```

As you'll recall, this function takes an array of treasures and a target value, and tries to select a subset of the treasures that total the target value exactly. If there is such a set, it returns an array with just those treasures; if not, it returns `undef`.

We saw in Chapter 1 that this function has the same problem as the `fib` function: It can be slow, because it repeats the same work over and over again. When trying to select treasures from `1 2 3 4 5 6 7 8 9 10` that total 53, `find_share` comes upon the same situation twice: It finds that $1+2+3+6 = 12$, and invokes `find_share(41, [7,8,9,10])`, which eventually returns undefined. Then later, it finds that $1 + 2 + 4 + 5 = 12$, and invokes `find_share(41, [7,8,9,10])` a second time. Clearly, this is a good opportunity to try some caching.

For even this simple case, memoization yields a speed-up of about 68%. For larger examples, such as `find_share(200, [1..20])`, the speedup is larger, about 82%. In some cases, memoization can make the difference between a practical and an impractical algorithm. The unmemoized version of `find_share([1..20], 210)` takes several *thousand* times longer to run than the memoized version. (I used the key generation function `sub {join "-", @{$_[1]}, $_[0]}`.)

3.7.5 Custom Key Generation for Impure Functions

Custom key generation can also be used to deal with certain kinds of functions that depend on information other than their arguments.

Let's consider a long-running network server program whose job is to sell some product, such as pizzas or weapons-grade plutonium. The cost of a pizza or a canister of plutonium includes the cost of delivery, which in turn depends on the current hour of the day and day of the week. Delivery late at night and on weekends is more expensive because fewer delivery persons are available and

nobody likes to work at 3 AM.[7] The server might contain a function something like this:

CODE LIBRARY
delivery-charge

```
sub delivery_charge {
  my ($quantity_ordered) = @_;
  my ($hour, $day_of_week) = (localtime)[2,6];
  # perform complex computation involving $weight, $gross_cost,
  #     $hour, $day_of_week, and $quantity_ordered
  # ...
  return $delivery_charge;
}
```

Because the function is complicated, we would like to memoize it. The default key generator, `join(',', @_)`, is unsuitable here, because we would lose the time dependence of the delivery charge. But it's easy to solve the problem with a custom key-generation function such as:

```
sub delivery_charge_key {
  join ',', @_, (localtime)[2,6];
}
```

`delivery_charge` is not a pure function, but in this case it may not matter. The only real issue is whether there will be enough cache hits to gain a performance win. We might expect the function to have many cache misses for the first week, until the day of the week rolls over, and then to start seeing more cache hits. In this case the effectiveness of the caching might depend on the longevity of the program. Similarly, we might wonder if the following key-generation function wouldn't be better:

```
sub delivery_charge_key {
  my ($hour, $day_of_week) = (localtime)[2,6];
  my $weekend = $day_of_week == 0 || $day_of_week == 6;
  join ',', @_, $hour, $weekend;
}
```

This function takes longer to run, but might get more cache hits because it recognizes that values cached on Monday may be used again on Tuesday and Wednesday. Again, which is best will depend on subtle factors in the program's behavior.

---

7 Most plutonium is ordered late at night in spite of the extra costs.

3.8 CACHING IN OBJECT METHODS

For object methods, it often makes little sense to store the cached values in a separate hash. Consider an `Investor` object in a program written by an investment bank. The object represents one of the bank's customers:

```
package Investor;

# Compute total amount currently invested
sub total {
  my $self = shift;
  # ... complex computation performed here ...
  return $total;
}
```

If the `$total` is not expected to change, we might add caching to it, using the object's identity as a key into the cache hash:

```
# Compute total amount currently invested
{ my %cache;
  sub total {
    my $self = shift;
    return $cache{$self} if exists $cache{$self};
    # ... complex computation performed here ...
    return $cache{$self} = $total;
  }
}
```

However, this technique has a serious problem. When we use an object as a hash key, Perl converts it to a string. The typical hash key will look like `Investor=HASH(0x80ef8dc)`. The hexadecimal numeral is the address at which the object's data is actually stored. It is essential that this key be different for every two objects, or we run the risk of false cache hits, where we retrieve one object's total while thinking that it belongs to a different object. In Perl, these hash keys are indeed distinct for all the live objects in the system at any given time, but no guarantee is made about dead objects. If an object is destroyed and a new object is created, the new object might very well exist at the same memory address formerly occupied by the old object and thus be confused with it:

```
# here 90,000 is returned from the cache
$old_total = $old_object->total();
```

```
undef $old_object;
$new_object = Investor->new();
$new_total = $new_object->total();
```

Here we ask for the total for the new investor. It should be 0, since the investor is new. But instead, the `->total` method happens to look in the cache under the same hash key that was used by the `$old_object` that was recently destroyed; the method sees the `90000` stored there, and returns it erroneously. This problem can be solved with a `DESTROY` method that deletes an object's data from the cache, or by associating a unique, non-re-usable ID number with every object in the program, and using the ID number as the hash key, but there is a much more straightforward solution.

In an OOP context, the cache hash technique is peculiar, because there is a more natural place to store the cached data: as member data in the object itself. A cached total becomes another property that might or might not be carried by each individual object:

```
# Compute total amount currently invested
sub total {
  my $self = shift;
  return $self->{cached_total} if exists $self->{cached_total};
  # ... complex computation performed here ...
  return $self->{cached_total} = $total;
}
```

Here the logic is exactly the same as before; the only difference is that the method stores the total for each object in the object itself, instead of in an auxiliary hash. This avoids the problem of hash key collision that arose with the auxiliary hash.

Another advantage of this technique is that the space devoted to storage of the cached total is automatically reclaimed when the object is destroyed. With the auxiliary hash, each cached value would persist forever, even after the object to which it belonged was destroyed.

Finally, storing the cached information in each object allows more flexible control over when it is expired. In our example `total` computes the total amount of money that a certain investor has invested. Caching this total may be appropriate since investors may not invest new money too frequently. But caching it forever is probably not appropriate. In this example, whenever an investor invests more money, we need some way of signalling the `total` function that the cached total is no longer correct, and must be recomputed from scratch. This is called *expiring* the cached value.

With the auxiliary hash technique, there was no way to do this without adding a special-purpose method in the scope of the cache hash, something like this:

```
# Compute total amount currently invested
{ my %cache;
  sub total {
    my $self = shift;
    return $cache{$self} if exists $cache{$self};
    # ... complex computation performed here ...
    return $cache{$self} = $total;
  }

  sub expire_total {
    my $self = shift;
    delete $cache{$self};
  }
}

sub invest {
  my ($self, $amount, ...) = @_;
  $self->expire_total;
  ...
}
```

With the object-oriented technique, no special method is necessary, because each method can directly expire the cached total if it needs to:

```
# Compute total amount currently invested
sub total {
  my $self = shift;
  return $self->{cached_total} if exists $self->{cached_total};
  # ... complex computation performed here ...
  return $self->{cached_total} = $total;
}

sub invest {
  my ($self, $amount, ...) = @_;
  delete $self->{cached_total};
  ...
}
```

3.8.1 Memoization of Object Methods

As we saw, for object methods, we often like to cache each computed value with the object for which it is relevant, instead of in one separate hash. The `memoize` function we saw earlier doesn't do this, but it's not hard to build one that does:

CODE LIBRARY
memoize-method

```
sub memoize_method {
  my ($method, $key) = @_;
  return sub {
    my $self = shift;
    return $self->{$key} if exists $self->{$key};
    return $self->{$key} = $method->($self, @_);
  };
}
```

`$method` is a reference to the true method. `$key` is the name of the slot in each object in which the cached values will be stored. The stub function returned here is suitable for use as a method. When the stub is invoked, it retrieves the object on whose behalf it was called, just like any other method; then it looks in the object for member data named `$key` to see if a value is cached there. If so, the stub returns the cached value; if not, it calls the real method, caches the result in the object, and returns the new cached result.

To use this, we might write something like this:

```
*Investor::total = memoize_method(\&Investor::total, 'cached_total');
$investor_bob->total;
```

This installs the stub in the symbol table in place of the original method. Alternatively, we might use:

```
$memoized_total = memoize_method(\&Investor::total, 'cached_total');
$investor_bob->$memoized_total;
```

These are not quite the same. In the former case, all calls to `->total` will use the memoized version of the method, including calls from subclasses that inherit the method. In the latter case, we get the memoized version of the method only when we explicitly ask for it by using the `->$memoized(...)` notation.

3.9 PERSISTENT CACHES

Before we leave the topic of automatic memoization, we'll see a few peripheral techniques. We saw how a function could be replaced with a memoized version that stored return values in a cache; the cache was simply a hash variable.

In Perl, one can use the `tie` operator to associate a hash variable with a disk database, so that data stored in the hash is automatically written to the disk, and data fetched back from the hash actually comes from the disk. To add this feature to our `memoize` function is simple:

```
use DB_File;

sub memoize {
  my ($func, $keygen, $file) = @_;
  my %cache;
  if (defined $file) {
    tie %cache => 'DB_File', $file, O_RDWR|O_CREAT, 0666
      or die "Couldn't access cache file $file: $!; aborting";
  }
  my $stub = sub {
    my $key = $keygen ? $keygen->(@_) : join ',', @_;
    $cache{$key} = $func->(@_) unless exists $cache{$key};
    return $cache{$key};
  };
  return $stub;
}
```

Here we've added an optional third parameter, which is the name of the disk file that will receive the cached data. If supplied, we use `tie` to tie the hash to the file. Note that if you don't use this feature, you pay hardly any cost at all — a single `defined()` test at the time you call `memoize()`.

When the cache hash is tied to a disk file in this way, the cache becomes persistent. Data stored in the cache on one run of the program remains in the file after the program has exited, and is available to the function the next time the program is run. The program is incrementally replacing the function with a lookup table on the disk. The cost to the programmer is nearly zero, since we did not have to change any of the code in the original function.

If we get tired of waiting for the lookup table to be completely populated, we can force the issue. We can write a tiny program that does nothing but call the memoized function over and over with different arguments each time. We start

it up on Friday afternoon and go home for the weekend. When we come back on Monday, the persistent cache will have the values of the function precomputed. When we run our real application, all calls to the memoized function will return almost instantly, since the values have been saved in the database.

Once again, this may not be a win. Remember, the speed-up from memoization is $hf - K$, where K is the overhead of managing the cache. If K is sufficiently large, it will overwhelm the gains from the hf part of the formula, as in the `sub { $_[0] * $_[0] }` example from Section 3.6. When we store cache data in a disk file, the overhead K can be many times greater than normal, because our program will have to make an operating-system request to look in the disk database.

An alternative and more flexible interface is to allow the user of `memoize()` to supply their own tied hash:

```
sub memoize {
    my ($func, $keygen, $cache) = @_;
    $cache = {} unless defined $cache;
    my $stub = sub {
       my $key = $keygen ? $keygen->(@_) : join ',', @_;
       $cache->{$key} = $func->(@_) unless exists $cache->{$key};
       return $cache->{$key};
    };
    return $stub;
}
```

This allows the user to supply a cache that is tied to a disk file using their favorite DBM implementation, even one we've never heard of. They could also pass in an ordinary hash; that would allow them to erase the cache or to expire old values from it if they wanted to.

3.10 ALTERNATIVES TO MEMOIZATION

Most pure functions present an opportunity for caching. Although it may appear at first that pure functions are rare, they do appear with some frequency. One place where pure functions are especially common is as the comparator functions used in sorting.

The Perl built-in `sort` operator is generic, in that it can sort a list of any kind of data into any order desired by the program. By default, it sorts a list of strings into alphabetical order, but the programmer may optionally supply a *comparator function* that tells Perl how to reorder `sort`'s argument list. The comparator

function is called repeatedly, each time with two different elements from the list to be sorted, and must return a negative value if the two elements are in the correct order, a positive value if the two elements are in the wrong order, and zero if it doesn't care. Typically, a comparator function's return value depends only on the values of its arguments, the two list items it is comparing, so it is a pure function.

Probably the simplest example of a comparator function is the comparator that compares numbers for sorting into numerical order:

```
@sorted_numbers = sort { $a <=> $b } @numbers;
```

Here `{ $a <=> $b }` is the comparator function. The `sort` operator examines the list of `@numbers`, sets `$a` and `$b` to the numbers it wishes to have compared, and invokes the comparator function. `<=>` is a special Perl operator that returns a negative value if `$a` is less than `$b`, a positive value if `$a` is greater than `$b`, and zero if `$a` and `$b` are equal.[8] `cmp` is an analogous operator for strings; this is the default that Perl uses if you don't specify an explicit comparator.

An alternative syntax uses a named function instead of a bare block:

```
@sorted_numbers = sort numerically @numbers;
sub numerically { $a <=> $b }
```

This is equivalent to the bare-block version.

A more interesting example sorts a list of date strings of the form `"Apr 16, 1945"` into chronological order:

CODE LIBRARY
chrono-1

```
@sorted_dates = sort chronologically @dates;

%m2n =
   ( jan =>  1, feb =>  2, mar =>  3,
     apr =>  4, may =>  5, jun =>  6,
     jul =>  7, aug =>  8, sep =>  9,
     oct => 10, nov => 11, dec => 12, );

sub chronologically {
  my ($am, $ad, $ay) =
    ($a =~ /(\w{3}) (\d+), (\d+)/);
```

8 Subtraction would work equally well here; <=> is used in comparators instead of plain subtraction because of its documentative value.

```
  my ($bm, $bd, $by) =
    ($b =~ /(\w{3}) (\d+), (\d+)/);

             $ay  <=>         $by
  || $m2n{lc $am} <=> $m2n{lc $bm}
  ||         $ad  <=>         $bd;
}
```

The two date strings to be compared are loaded into `$a` and `$b`, as before, and then split up into `$ay`, `$by`, `$am`, and so forth. `$ay` and `$by`, the years, are compared first. The `||` operator here is a common idiom in sort comparators for sorting by secondary keys. The `||` operator returns its left operand, unless that is zero, in which case it returns its right operand. If the years are the same, then `$ay <=> $by` returns zero, and the `||` operator passes control to the part of the expression involving the months, which are used to break the tie. But if the years are different, then the result of the first `<=>` is nonzero, and this is the result of the `||` expression, instructing `sort` how to order `$a` and `$b` in the result list without ever having looked at the months or the days. If control passes to the `$am <=> $bm` part, the same thing happens. The months are compared; if the result is conclusive, the function returns immediately, and if the months are the same, control passes to the final tiebreaker of comparing the days.

Internally, Perl's `sort` operator has been implemented with various algorithms that have $O(n \log n)$ running time. This means that to sort a list that is n times larger than another typically takes somewhat more than n times as long. If the list size doubles, the running time more than doubles. The following table compares the length of the argument list with the number of calls typically made to the comparator function:

Length	# calls	calls / element
5	7	1.40
10	26	2.60
20	60	3.00
40	195	4.87
80	417	5.21
100	569	5.69
1000	9502	9.50
10000	139136	13.91

I got the "# calls" column by generating a list of random numbers of the indicated length and sorting it with a comparator function that incremented a counter each time it was called. The number of calls will vary depending on the list and on the comparator function, but these values are typical.

Now consider a list of 10,000 dates. 139,136 calls are made to the comparator function; each call performs two pattern-match operations, so there are 278,272 pattern matches in all. This means each date is split up into year, month, and day 27.8 times on average. Since the three components for a given date never change, it's clear that 26.8 of these matchings are wasted.

The first thing that might come to mind is to memoize the `chronologically` function, but this doesn't work well in practice. Although `sort` will call `chronologically` repeatedly with the same date, it won't call it twice on the same *pair* of dates (Unless, of course, the input list contains duplicate dates.) Since the hash keys must incorporate both arguments, the memoized function will never have a cache hit.

Instead, we'll do something slightly different, and memoize just the expensive part of the function. This will require a version of `memoize()` that can handle a function that returns a list.

CODE LIBRARY
chrono-2

```
@sorted_dates = sort chronologically @dates;

%m2n =
   ( jan =>  1, feb =>  2, mar =>  3,
     apr =>  4, may =>  5, jun =>  6,
     jul =>  7, aug =>  8, sep =>  9,
     oct => 10, nov => 11, dec => 12, );

sub chronologically {
  my ($am, $ad, $ay) = split_date($a);
  my ($bm, $bd, $by) = split_date($b);

             $ay  <=>          $by
  || $m2n{lc $am} <=> $m2n{lc $bm}
  ||          $ad  <=>          $bd;
}

sub split_date {
  $_[0] =~ /(\w{3}) (\d+), (\d+)/;
}
```

If we set up caching on `split_date`, we'll still make 278,272 calls to it, but 268,272 will result in cache hits, and only the remaining 10,000 will require pattern matching. The only catch is that we'll have to write the caching code by hand, because `split_date` returns a list, and our `memoize` functions deal correctly only with functions that return scalars.

At this point, we could go in three directions. We could enhance our `memoize` function to deal correctly with list-context returns. (Or we could use the CPAN

`Memoize` module, which does work correctly for functions that return lists.) We could write the caching code manually. But it's more instructive to sidestep the problem by replacing `split_date` with a function that returns a scalar. If the scalar is constructed correctly, we will be able to dispense with the complicated `||` logic in `chronologically` and just use a simple string compare.

Here's the idea: We will split the date, as before, but instead of returning a list of fields, we will pack the fields into a single string. The fields will appear in the string in the order we need to examine them, with the year first, then the month, then the day. The string for `"Apr 16, 1945"` will be `"19450416"`. When we compare strings with `cmp`, Perl will stop comparing as soon as possible, so if one string begins with `"1998..."` and another with `"1996..."` Perl will know the result as soon as it sees the fourth character, and won't bother to examine the month or day. String comparison is very fast, likely to beat out a sequence of `<=>`s and `||`s.

Here's the modified code:

CODE LIBRARY
chrono-3

```
@sorted_dates = sort chronologically @dates;

%m2n =
   ( jan =>  1, feb =>  2, mar =>  3,
     apr =>  4, may =>  5, jun =>  6,
     jul =>  7, aug =>  8, sep =>  9,
     oct => 10, nov => 11, dec => 12, );

sub chronologically {
  date_to_string($a) cmp date_to_string($b)
}

sub date_to_string {
  my ($m, $d, $y) = ($_[0] =~ /(\w{3}) (\d+), (\d+)/);
  sprintf "%04d%02d%02d", $y, $m2n{lc $m}, $d;
}
```

Now we can memoize `date_to_string`. Whether this will win over the previous version depends on whether the `sprintf` plus `cmp` is faster than the `<=>` plus `||`. As usual, a benchmark is required; it turns out that the code with the `sprintf` is about twice as fast.[9]

9 This comes as a surprise to many people, especially C programmers who expect `sprintf` to be slow. While `sprintf` is slow, so is Perl, so that dispatching a bunch of extra `<=>` and `||` operations

Sorting is often one of those places in the program where we need to squeeze out as much performance as possible. For a list of 10,000 dates, we call `sprintf` exactly 10,000 times (once `date_to_string` is memoized) but we still call `date_to_string` itself 278,272 times. As the list of dates becomes longer, this disparity will increase, and the function calls will eventually come to dominate the running time of the sort.

We can get more speed by simplifying the cache handling and eliminating the 268,272 extra function calls. To do this, we go back to handwritten caching code:

CODE LIBRARY
chrono-orc

```
{ my %cache;
  sub chronologically {
    ($cache{$a} ||= date_to_string($a))
       cmp
    ($cache{$b} ||= date_to_string($b))
  }
}
```

Here we make use of the `||=` operator, which seems almost custom-made for caching applications. `$x ||= $y` yields the value of `$x` if it is true; if not, it assigns `$y` to `$x` and yields the value of `$y`. `$cache{$a} ||= date_to_string($a)` checks to see if `$cache{$a}` has a true value; if so, that is the value used in the comparison with the `cmp` operator. If nothing is cached yet, then `$cache{$a}` is false, and `chronologically` calls `date_to_string`, stores the result in the cache, and uses the result in the comparison. This inline cache technique is called the *Orcish Maneuver*, because its essential features are the `||` and the cache.[10]

Memoizing `date_to_string` yields a two-and-a-half-fold speed-up; replacing the memoization with the Orcish Maneuver yields an *additional* twofold speed-up.

Astute readers will note that the Orcish Maneuver doesn't always work quite right. In this example, it's impossible for `date_to_string` to ever return a false value. But let's return for a moment to the example where we compute the total amount invested for each investor:

```
{ my %cache;
  sub by_total_invested {
    ($cache{$a} ||= total_invested($a))
```

takes a long time compared to `sprintf`. This is just another example of why the benchmark really is necessary.

10 Joseph Hall, author of *Effective Perl Programming*, is responsible for this name.

```
        <=>
    ($cache{$b} ||= total_invested($b))
  }
}
```

Suppose Luke the Hermit has invested no money at all. The first time he appears in `by_total_invested`, we call `total_invested` for Luke, and we get back 0. We store this 0 in the cache under Luke's key. The next time Luke appears, we check the cache and find that the value stored there is 0. Because this value is false, we call `total_invested` again, even though we had a cache hit. The problem here is that the `||=` operator doesn't distinguish between a cache miss and a cache hit where the cached value happens to be false.

The Lisp people have a name for this phenomenon: They call it the *semipredicate problem*. A *predicate* is a function that returns a boolean value. A *semipredicate* can return a specific false value, indicating failure, or one of many meaningful true values, indicating success. The `$cache{$a}` is a semipredicate because it might return 0, or any of an infinity of useful true values. We get into trouble when 0 is *also* one of the true values, because we can't distinguish it from the 0 that means false. This is the semipredicate problem.

In our present example, the semipredicate problem won't cause much trouble. The only cost is a few extra calls to `total_invested` for people who haven't invested any money. If we find that these extra calls are slowing down our sorting significantly (unlikely, but possible) we can replace the comparator function with the following version:

```
{ my %cache;
  sub by_total_invested {
   (exists $cache{$a} ? $cache{$a} : ($cache{$a} = total_invested($a)))
       <=>
   (exists $cache{$b} ? $cache{$b} : ($cache{$b} = total_invested($b)))
  }
}
```

This version uses the reliable `exists` operator to check to see if the cache is populated. Even if the value stored in the cache is false, `exists` will still return true. Beware, though, that this costs about 10% more than the simpler version.

There's an alternative that costs hardly anything extra, but does have the disadvantage of being rather bizarre. It's based on the following trick: When the Perl string `"0e0"` is used as a number, it behaves exactly like 0; the `e0` is interpreted

by Perl as a scientific notation exponent. But unlike an ordinary 0, the string `"0e0"` is true rather than false.[11]

If we write `by_total_invested` like this, we avoid the semipredicate problem with hardly any extra cost:

```
{ my %cache;
  sub by_total_invested {
    ($cache{$a} ||= total_invested($a) || "0e0")
       <=>
    ($cache{$b} ||= total_invested($b) || "0e0")
  }
}
```

If `total_invested` returns zero, the function caches `"0e0"` instead. The next time we look up the total invested by the same customer, the function sees `"0e0"` in the cache, and this value is true, so it doesn't call `total_invested` a second time. This `"0e0"` is the value given to the `<=>` operator for comparison, but in a numeric comparison it behaves exactly like 0, which is just what we want. The speed cost of the additional `||` operation, invoked only when a false value is returned by `total_invested()`, is tiny.

3.11 EVANGELISM

If you're trying to explain to a C programmer why Perl is good, automatic memoization makes a wonderful example. Almost all programmers are familiar with caching techniques. Even if they don't use any caching techniques in their own programs, they are certainly familiar with the concept, from caching in web browsers, in the cache memory of their computer, in the DNS server, in their web proxy server, or elsewhere. Caching, like most simple, useful ideas, is ubiquitous.

Adding caching isn't too much trouble, but it takes at least a few minutes to modify the code. With all modifications, there's a chance that you might make a mistake, which has to be factored into the average time. Once you're done, it

11 `"0e0"` is hardly unique; `"00"` will also work, as will any string that begins with a 0 followed by a non-numeral character, such as `"0!!!!"`. Strings like `"0!!!!"`, however, will generate an "Argument isn't numeric" warning if warnings are enabled. One string commonly used when a zero-but-true value is desired is `"0 but true"`. Perl's warning system has a special case in it that suppresses the usual "isn't numeric" warning for this string.

may turn out that the caching was a bad idea, because the cache management overhead dominates the running time of the function, or because there aren't as many cache hits on a typical run as you expected there to be; then you have to take the caching code out, and again you run the risk of making a mistake. Not to overstate the problems, of course, but it will take at least a few minutes in each direction.

With memoization, adding the caching code no longer takes minutes; it takes seconds. You add one line of code:

```
memoize 'myfunction';
```

and it is impossible to make a serious mistake and break the function. If the memoization turns out to have been a bad idea, you can turn it off again in *one second*. Most programmers can appreciate the convenience of this. If you have five minutes to explain to a C programmer what benefits Perl offers over C, memoization is an excellent example to use.

3.12 THE BENEFITS OF SPEED

It may be tempting at this point to say that memoization is only an incremental improvement over manual caching techniques, because it does the same thing, and the only additional benefit is that you can turn it on and off more quickly. But that isn't really true. When a speed and convenience difference between tools is large enough, it changes the way you think about the tool and the ways you can use it. To automatically memoize a function takes 1/100 as much time as to write caching code for it manually. This is the same as the difference between the speed of an airplane and the speed of an oxcart. To say that the airplane is just a faster oxcart is to miss something essential: The quantitative difference is so large that it becomes a substantive qualitative difference as well.

For example, with automatic memoization, it becomes possible to add caching behavior to functions without having to consider the performance details carefully in advance. Memoization is so easy that it can pay to adopt a strategy of "shoot first and ask questions later." If a function is slow, try slapping some caching onto it and see if it helps matters. If a recursive function might have bad recursion behavior, put in some caching and see if the problem goes away. If not, you can take the caching away again and investigate more thoroughly. When the total cost is ten seconds of programming time, you can try this without having to think much in advance about whether it will be successful. With manual caching, you would have to spend at least a quarter hour, which is too much to invest on a mere fishing expedition.

With automatic memoization, you can enable caching behavior at run time. For example:

```
sub function {
  if (++$CALLS == 100) { memoize 'function'}
  ...
}
```

Here we don't bother to memoize the function until partway through the program's run. When the function realizes it's being heavily used, it enables caching behavior. To do the same thing without automatic memoization requires a rewrite of the function rather than the addition of a single line.

3.12.1 Profiling and Performance Analysis

Automatic memoization allows caching to be used in profiling and performance analysis in a way that would be impractical otherwise. The typical situation involves a large application that runs too slowly. We would like to speed it up. We will do this by rewriting parts of the program to be faster, perhaps by introducing a better algorithm, and possibly at the expense of a certain amount of clarity or maintainability.

Trying to speed up every part of the program is a bad allocation of resources. This is because of what is known as the"90-10 rule," which says that 90% of the execution time of a program takes place in only 10% of the code, the rest being initialization code that is executed only once, or special-case code such as error handlers that are executed rarely or never. If we work over the entire program and speed up every part by 5%, we have a 5% gain. But if we can identify and rewrite just the crucial 10% to the same degree, we will get a net 4.5% gain in the program's total run time at only 10% of the cost; the cost-benefit ratio is nine times as large. So before we optimize, we would like very much to identify the parts of the program that contribute most to the run time, and concentrate on improving just those parts.

It's sad when a programmer spends a week carefully optimizing a subroutine to run 20% faster, only to discover that the program spent only 2% of its total execution time in that subroutine, and that the week of hard work has yielded only an 0.4% speed-up overall. Historically, programmers have been bad at guessing which parts of the program are heavily used; we need real measurements.

Traditionally, measurements are done using a tool called a *profiler*. The program is run in a special profiling environment that causes it to dump out a record of what it is doing every so often (typically many times per second).

Afterwards, the data is massaged into a report that lists the subroutines in which the program spends the most execution time. There are profiler tools for Perl, but they can be strange and hard to use. Automatic memoization is an alternative.

Run the program once and time how long it takes. Then guess which parts of the program are bottlenecks, and memoize them. Arrange for the memoized data to be stored in a persistent file. (Remember, this requires adding only one line of code to the program.) Run the program a second time; this will populate the cache on the disk. Run the program a third time. All calls to the memoized functions will return almost immediately, because the data is residing in a disk database; the functions do no work at all beyond what is required to get the answers from the database. On the third run, you are simulating how quickly the program would run if it were possible to eliminate the time taken by the target functions. If this run is substantially faster than the unmemoized run time, you have some candidates for optimization; if the times are similar, you know that you should look elsewhere.

You might wonder why not simply leave the memoization in place if the memoized run is substantially shorter, and the answer is that while memoization might cause the target functions to run faster, it might also cause them not to work correctly. Suppose you suspect that the bottleneck function is the one that formats the report. While memoizing this function and having it deliver a precached report may cause it to run faster, it is probably not what the recipient of the report would prefer.

3.12.2 Automatic Profiling

Another profiling technique, one that's even more flexible, uses the techniques we've seen in this chapter, but without any actual caching. The `memoize` function takes an existing function and puts a cache-managing front-end onto it. There's no reason why this front-end has to do cache management; it could do something else:

CODE LIBRARY
profile

```
use Time::HiRes 'time';
my (%time, %calls);

sub profile {
  my ($func, $name) = @_;
  my $stub = sub {
    my $start = time;
    my $return = $func->(@_);
    my $end = time;
```

```
    my $elapsed = $end - $start;
    $calls{$name} += 1;
    $time{$name} += $elapsed;
    return $return;
  };
  return $stub;
}
```

The `profile` function shown here is similar in structure to the `memoize` function we saw earlier. Like `memoize`, it takes a function as its argument and constructs and returns a stub, which may be called directly or installed in the symbol table in place of the original.

When the stub is invoked, it records the current time in `$start`. Normally the Perl `time` function returns the current time to the nearest second; the `Time::HiRes` module replaces the `time` function with one that has finer granularity, if possible. The stub calls the real function and saves its return value. Then it computes the total elapsed time and updates two hashes. One hash records, for each function, how many calls have been made to that function; the stub simply increments that count. The other hash records the total elapsed time spent executing each function; the stub adds the elapsed time for the just-completed call to the total.

At the end of program execution, we can print out a report:

```
END {
  printf STDERR "%-12s %9s %6s\n", "Function", "# calls", "Elapsed";
  for my $name (sort {$time{$b} <=> $time{$a}} (keys %time)) {
    printf STDERR "%-12s %9d %6.2f\n", $name, $calls{$name}, $time{$name};
  }
}
```

The output will look something like this:

Function	# calls	Elapsed
`printout`	1	10.21
`searchfor`	1	0.34
`page`	1	0.06
`check_file`	18	0.01

This is output from the `perldoc` program that comes standard with Perl. From this output, we can see that most of the execution time is occurring in the `printout` function; if we want to make `perldoc` faster, this is the function we should concentrate on.

3.12.3 Hooks

This is clearly a very rudimentary profiling tool. A better version would use the `times()` function to measure CPU time consumed instead of wall-clock time. But the flexibility of the technique should be clear; we can put an arbitrary front-end onto a function, or change the front-end at run time. The front-end can perform caching, or keep track of function call data; it could validate the function arguments if we wanted, enforce pre- and post-conditions, or whatever else we like.