NAME

perlpacktut - tutorial on `pack` and `unpack`

DESCRIPTION

`pack` and `unpack` are two functions for transforming data according to a user-defined template, between the guarded way Perl stores values and some well-defined representation as might be required in the environment of a Perl program. Unfortunately, they're also two of the most misunderstood and most often overlooked functions that Perl provides. This tutorial will demystify them for you.

The Basic Principle

Most programming languages don't shelter the memory where variables are stored. In C, for instance, you can take the address of some variable, and the `sizeof` operator tells you how many bytes are allocated to the variable. Using the address and the size, you may access the storage to your heart's content.

In Perl, you just can't access memory at random, but the structural and representational conversion provided by `pack` and `unpack` is an excellent alternative. The `pack` function converts values to a byte sequence containing representations according to a given specification, the so-called "template" argument. `unpack` is the reverse process, deriving some values from the contents of a string of bytes. (Be cautioned, however, that not all that has been packed together can be neatly unpacked - a very common experience as seasoned travellers are likely to confirm.)

Why, you may ask, would you need a chunk of memory containing some values in binary representation? One good reason is input and output accessing some file, a device, or a network connection, whereby this binary representation is either forced on you or will give you some benefit in processing. Another cause is passing data to some system call that is not available as a Perl function: `syscall` requires you to provide parameters stored in the way it happens in a C program. Even text processing (as shown in the next section) may be simplified with judicious usage of these two functions.

To see how (un)packing works, we'll start with a simple template code where the conversion is in low gear: between the contents of a byte sequence and a string of hexadecimal digits. Let's use `unpack`, since this is likely to remind you of a dump program, or some desperate last message unfortunate programs are wont to throw at you before they expire into the wild blue yonder. Assuming that the variable `$mem` holds a sequence of bytes that we'd like to inspect without assuming anything about its meaning, we can write

```
my( $hex ) = unpack( 'H*', $mem );
print "$hex\n";
```

whereupon we might see something like this, with each pair of hex digits corresponding to a byte:

```
41204d414e204120504c414e20412043414e414c2050414e414d41
```

What was in this chunk of memory? Numbers, characters, or a mixture of both? Assuming that we're on a computer where ASCII (or some similar) encoding is used: hexadecimal values in the range `0x40` - `0x5A` indicate an uppercase letter, and `0x20` encodes a space. So we might assume it is a piece of text, which some are able to read like a tabloid; but others will have to get hold of an ASCII table and relive that firstgrader feeling. Not caring too much about which way to read this, we note that `unpack` with the template code `H` converts the contents of a sequence of bytes into the customary hexadecimal notation. Since "a sequence of" is a pretty vague indication of quantity, `H` has been defined to convert just a single hexadecimal digit unless it is followed by a repeat count. An asterisk for the repeat count means to use whatever remains.

The inverse operation - packing byte contents from a string of hexadecimal digits - is just as easily written. For instance:

```
my $s = pack( 'H2' x 10, map { "3$_" } ( 0..9 ) );
print "$s\n";
```

Since we feed a list of ten 2-digit hexadecimal strings to `pack`, the pack template should contain ten pack codes. If this is run on a computer with ASCII character coding, it will print `0123456789`.

Packing Text

Let's suppose you've got to read in a data file like this:

```
Date      |Description               | Income|Expenditure
01/24/2001 Ahmed's Camel Emporium               1147.99
01/28/2001 Flea spray                             24.99
01/29/2001 Camel rides to tourists      235.00
```

How do we do it? You might think first to use `split`; however, since `split` collapses blank fields, you'll never know whether a record was income or expenditure. Oops. Well, you could always use `substr`:

```
while (<>) {
    my $date   = substr($_,  0, 11);
    my $desc   = substr($_, 12, 27);
    my $income = substr($_, 40,  7);
    my $expend = substr($_, 52,  7);
    ...
}
```

It's not really a barrel of laughs, is it? In fact, it's worse than it may seem; the eagle-eyed may notice that the first field should only be 10 characters wide, and the error has propagated right through the other numbers - which we've had to count by hand. So it's error-prone as well as horribly unfriendly.

Or maybe we could use regular expressions:

```
while (<>) {
    my($date, $desc, $income, $expend) =
        m|(\d\d/\d\d/\d{4}) (.{27}) (.{7})(.*)|;
    ...
}
```

Urgh. Well, it's a bit better, but - well, would you want to maintain that?

Hey, isn't Perl supposed to make this sort of thing easy? Well, it does, if you use the right tools. `pack` and `unpack` are designed to help you out when dealing with fixed-width data like the above. Let's have a look at a solution with `unpack`:

```
while (<>) {
    my($date, $desc, $income, $expend) = unpack("A10xA27xA7A*", $_);
    ...
}
```

That looks a bit nicer; but we've got to take apart that weird template. Where did I pull that out of?

OK, let's have a look at some of our data again; in fact, we'll include the headers, and a handy ruler so we can keep track of where we are.

```
         1         2         3         4         5
12345678901234567890123456789012345678901234567890123456789012345678
Date      |Description               | Income|Expenditure
```

```
01/28/2001 Flea spray                                 24.99
01/29/2001 Camel rides to tourists      235.00
```

From this, we can see that the date column stretches from column 1 to column 10 - ten characters wide. The `pack`-ese for "character" is `A`, and ten of them are `A10`. So if we just wanted to extract the dates, we could say this:

```
my($date) = unpack("A10", $_);
```

OK, what's next? Between the date and the description is a blank column; we want to skip over that. The `x` template means "skip forward", so we want one of those. Next, we have another batch of characters, from 12 to 38. That's 27 more characters, hence `A27`. (Don't make the fencepost error - there are 27 characters between 12 and 38, not 26. Count 'em!)

Now we skip another character and pick up the next 7 characters:

```
my($date,$description,$income) = unpack("A10xA27xA7", $_);
```

Now comes the clever bit. Lines in our ledger which are just income and not expenditure might end at column 46. Hence, we don't want to tell our `unpack` pattern that we **need** to find another 12 characters; we'll just say "if there's anything left, take it". As you might guess from regular expressions, that's what the `*` means: "use everything remaining".

- Be warned, though, that unlike regular expressions, if the `unpack` template doesn't match the incoming data, Perl will scream and die.

Hence, putting it all together:

```
my($date,$description,$income,$expend) = unpack("A10xA27xA7xA*", $_);
```

Now, that's our data parsed. I suppose what we might want to do now is total up our income and expenditure, and add another line to the end of our ledger - in the same format - saying how much we've brought in and how much we've spent:

```
while (<>) {
    my($date, $desc, $income, $expend) = unpack("A10xA27xA7xA*", $_);
    $tot_income += $income;
    $tot_expend += $expend;
}

$tot_income = sprintf("%.2f", $tot_income); # Get them into
$tot_expend = sprintf("%.2f", $tot_expend); # "financial" format

$date = POSIX::strftime("%m/%d/%Y", localtime);

# OK, let's go:

print pack("A10xA27xA7xA*", $date, "Totals", $tot_income, $tot_expend);
```

Oh, hmm. That didn't quite work. Let's see what happened:

```
01/24/2001 Ahmed's Camel Emporium                   1147.99
01/28/2001 Flea spray                                 24.99
01/29/2001 Camel rides to tourists     1235.00
03/23/2001Totals                     1235.001172.98
```

OK, it's a start, but what happened to the spaces? We put `x`, didn't we? Shouldn't it skip forward? Let's look at what *"pack" in perlfunc* says:

```
x   A null byte.
```

Urgh. No wonder. There's a big difference between "a null byte", character zero, and "a space", character 32. Perl's put something between the date and the description - but unfortunately, we can't see it!

What we actually need to do is expand the width of the fields. The `A` format pads any non-existent characters with spaces, so we can use the additional spaces to line up our fields, like this:

```
print pack("A11 A28 A8 A*", $date, "Totals", $tot_income, $tot_expend);
```

(Note that you can put spaces in the template to make it more readable, but they don't translate to spaces in the output.) Here's what we got this time:

```
01/24/2001 Ahmed's Camel Emporium                  1147.99
01/28/2001 Flea spray                                24.99
01/29/2001 Camel rides to tourists     1235.00
03/23/2001 Totals                      1235.00 1172.98
```

That's a bit better, but we still have that last column which needs to be moved further over. There's an easy way to fix this up: unfortunately, we can't get `pack` to right-justify our fields, but we can get `sprintf` to do it:

```
$tot_income = sprintf("%.2f", $tot_income);
$tot_expend = sprintf("%12.2f", $tot_expend);
$date = POSIX::strftime("%m/%d/%Y", localtime);
print pack("A11 A28 A8 A*", $date, "Totals", $tot_income, $tot_expend);
```

This time we get the right answer:

```
01/28/2001 Flea spray                                24.99
01/29/2001 Camel rides to tourists     1235.00
03/23/2001 Totals                      1235.00      1172.98
```

So that's how we consume and produce fixed-width data. Let's recap what we've seen of `pack` and `unpack` so far:

- Use `pack` to go from several pieces of data to one fixed-width version; use `unpack` to turn a fixed-width-format string into several pieces of data.
- The pack format `A` means "any character"; if you're `pack`ing and you've run out of things to pack, `pack` will fill the rest up with spaces.
- `x` means "skip a byte" when `unpack`ing; when `pack`ing, it means "introduce a null byte" - that's probably not what you mean if you're dealing with plain text.
- You can follow the formats with numbers to say how many characters should be affected by that format: `A12` means "take 12 characters"; `x6` means "skip 6 bytes" or "character 0, 6 times".
- Instead of a number, you can use `*` to mean "consume everything else left".

  Warning: when packing multiple pieces of data, `*` only means "consume all of the current piece of data". That's to say

  ```
  pack("A*A*", $one, $two)
  ```

packs all of $one into the first A* and then all of $two into the second. This is a general principle: each format character corresponds to one piece of data to be packed.

Packing Numbers

So much for textual data. Let's get onto the meaty stuff that pack and unpack are best at: handling binary formats for numbers. There is, of course, not just one binary format - life would be too simple - but Perl will do all the finicky labor for you.

Integers

Packing and unpacking numbers implies conversion to and from some *specific* binary representation. Leaving floating point numbers aside for the moment, the salient properties of any such representation are:

- the number of bytes used for storing the integer,
- whether the contents are interpreted as a signed or unsigned number,
- the byte ordering: whether the first byte is the least or most significant byte (or: little-endian or big-endian, respectively).

So, for instance, to pack 20302 to a signed 16 bit integer in your computer's representation you write

```
my $ps = pack( 's', 20302 );
```

Again, the result is a string, now containing 2 bytes. If you print this string (which is, generally, not recommended) you might see 0N or N0 (depending on your system's byte ordering) - or something entirely different if your computer doesn't use ASCII character encoding. Unpacking $ps with the same template returns the original integer value:

```
my( $s ) = unpack( 's', $ps );
```

This is true for all numeric template codes. But don't expect miracles: if the packed value exceeds the allotted byte capacity, high order bits are silently discarded, and unpack certainly won't be able to pull them back out of some magic hat. And, when you pack using a signed template code such as s, an excess value may result in the sign bit getting set, and unpacking this will smartly return a negative value.

16 bits won't get you too far with integers, but there is l and L for signed and unsigned 32-bit integers. And if this is not enough and your system supports 64 bit integers you can push the limits much closer to infinity with pack codes q and Q. A notable exception is provided by pack codes i and I for signed and unsigned integers of the "local custom" variety: Such an integer will take up as many bytes as a local C compiler returns for sizeof(int), but it'll use *at least* 32 bits.

Each of the integer pack codes sSlLqQ results in a fixed number of bytes, no matter where you execute your program. This may be useful for some applications, but it does not provide for a portable way to pass data structures between Perl and C programs (bound to happen when you call XS extensions or the Perl function syscall), or when you read or write binary files. What you'll need in this case are template codes that depend on what your local C compiler compiles when you code short or unsigned long, for instance. These codes and their corresponding byte lengths are shown in the table below. Since the C standard leaves much leeway with respect to the relative sizes of these data types, actual values may vary, and that's why the values are given as expressions in C and Perl. (If you'd like to use values from %Config in your program you have to import it with use Config.)

```
signed unsigned  byte length in C   byte length in Perl
  s!     S!      sizeof(short)      $Config{shortsize}
  i!     I!      sizeof(int)        $Config{intsize}
  l!     L!      sizeof(long)       $Config{longsize}
```

```
    q!      Q!      sizeof(long long)  $Config{longlongsize}
```

The `i!` and `I!` codes aren't different from `i` and `I`; they are tolerated for completeness' sake.

Unpacking a Stack Frame

Requesting a particular byte ordering may be necessary when you work with binary data coming from some specific architecture whereas your program could run on a totally different system. As an example, assume you have 24 bytes containing a stack frame as it happens on an Intel 8086:

```
     +---------+        +----+----+               +---------+
TOS: |   IP    |  TOS+4:| FL | FH | FLAGS  TOS+14:|   SI    |
     +---------+        +----+----+               +---------+
     |   CS    |        | AL | AH | AX            |   DI    |
     +---------+        +----+----+               +---------+
                        | BL | BH | BX            |   BP    |
                        +----+----+               +---------+
                        | CL | CH | CX            |   DS    |
                        +----+----+               +---------+
                        | DL | DH | DX            |   ES    |
                        +----+----+               +---------+
```

First, we note that this time-honored 16-bit CPU uses little-endian order, and that's why the low order byte is stored at the lower address. To unpack such a (unsigned) short we'll have to use code `v`. A repeat count unpacks all 12 shorts:

```
my( $ip, $cs, $flags, $ax, $bx, $cd, $dx, $si, $di, $bp, $ds, $es ) =
  unpack( 'v12', $frame );
```

Alternatively, we could have used `C` to unpack the individually accessible byte registers FL, FH, AL, AH, etc.:

```
my( $fl, $fh, $al, $ah, $bl, $bh, $cl, $ch, $dl, $dh ) =
  unpack( 'C10', substr( $frame, 4, 10 ) );
```

It would be nice if we could do this in one fell swoop: unpack a short, back up a little, and then unpack 2 bytes. Since Perl *is* nice, it proffers the template code `X` to back up one byte. Putting this all together, we may now write:

```
my( $ip, $cs,
    $flags,$fl,$fh,
    $ax,$al,$ah, $bx,$bl,$bh, $cx,$cl,$ch, $dx,$dl,$dh,
    $si, $di, $bp, $ds, $es ) =
unpack( 'v2' . ('vXXCC' x 5) . 'v5', $frame );
```

(The clumsy construction of the template can be avoided - just read on!)

We've taken some pains to construct the template so that it matches the contents of our frame buffer. Otherwise we'd either get undefined values, or `unpack` could not unpack all. If `pack` runs out of items, it will supply null strings (which are coerced into zeroes whenever the pack code says so).

How to Eat an Egg on a Net

The pack code for big-endian (high order byte at the lowest address) is `n` for 16 bit and `N` for 32 bit integers. You use these codes if you know that your data comes from a compliant architecture, but, surprisingly enough, you should also use these pack codes if you exchange binary data, across the network, with some system that you know next to nothing about. The simple reason is that this order has been chosen as the *network order*, and all standard-fearing programs ought to follow this

convention. (This is, of course, a stern backing for one of the Lilliputian parties and may well influence the political development there.) So, if the protocol expects you to send a message by sending the length first, followed by just so many bytes, you could write:

```
my $buf = pack( 'N', length( $msg ) ) . $msg;
```

or even:

```
my $buf = pack( 'NA*', length( $msg ), $msg );
```

and pass `$buf` to your send routine. Some protocols demand that the count should include the length of the count itself: then just add 4 to the data length. (But make sure to read *Lengths and Widths* before you really code this!)

Byte-order modifiers

In the previous sections we've learned how to use `n`, `N`, `v` and `V` to pack and unpack integers with big- or little-endian byte-order. While this is nice, it's still rather limited because it leaves out all kinds of signed integers as well as 64-bit integers. For example, if you wanted to unpack a sequence of signed big-endian 16-bit integers in a platform-independent way, you would have to write:

```
my @data = unpack 's*', pack 'S*', unpack 'n*', $buf;
```

This is ugly. As of Perl 5.9.2, there's a much nicer way to express your desire for a certain byte-order: the > and < modifiers. > is the big-endian modifier, while < is the little-endian modifier. Using them, we could rewrite the above code as:

```
my @data = unpack 's>*', $buf;
```

As you can see, the "big end" of the arrow touches the `s`, which is a nice way to remember that > is the big-endian modifier. The same obviously works for <, where the "little end" touches the code.

You will probably find these modifiers even more useful if you have to deal with big- or little-endian C structures. Be sure to read *Packing and Unpacking C Structures* for more on that.

Floating point Numbers

For packing floating point numbers you have the choice between the pack codes `f`, `d`, `F` and `D`. `f` and `d` pack into (or unpack from) single-precision or double-precision representation as it is provided by your system. If your systems supports it, `D` can be used to pack and unpack extended-precision floating point values (`long double`), which can offer even more resolution than `f` or `d`. `F` packs an `NV`, which is the floating point type used by Perl internally. (There is no such thing as a network representation for reals, so if you want to send your real numbers across computer boundaries, you'd better stick to ASCII representation, unless you're absolutely sure what's on the other end of the line. For the even more adventuresome, you can use the byte-order modifiers from the previous section also on floating point codes.)

Exotic Templates

Bit Strings

Bits are the atoms in the memory world. Access to individual bits may have to be used either as a last resort or because it is the most convenient way to handle your data. Bit string (un)packing converts between strings containing a series of `0` and `1` characters and a sequence of bytes each containing a group of 8 bits. This is almost as simple as it sounds, except that there are two ways the contents of a byte may be written as a bit string. Let's have a look at an annotated byte:

```
  7 6 5 4 3 2 1 0
+-----------------+
| 1 0 0 0 1 1 0 0 |
```

```
+-----------------+
 MSB           LSB
```

It's egg-eating all over again: Some think that as a bit string this should be written "10001100" i.e. beginning with the most significant bit, others insist on "00110001". Well, Perl isn't biased, so that's why we have two bit string codes:

```
$byte = pack( 'B8', '10001100' ); # start with MSB
$byte = pack( 'b8', '00110001' ); # start with LSB
```

It is not possible to pack or unpack bit fields - just integral bytes. `pack` always starts at the next byte boundary and "rounds up" to the next multiple of 8 by adding zero bits as required. (If you do want bit fields, there is *"vec" in perlfunc*. Or you could implement bit field handling at the character string level, using split, substr, and concatenation on unpacked bit strings.)

To illustrate unpacking for bit strings, we'll decompose a simple status register (a "-" stands for a "reserved" bit):

```
+-----------------+-----------------+
| S Z - A - P - C | - - - - O D I T |
+-----------------+-----------------+
 MSB           LSB MSB           LSB
```

Converting these two bytes to a string can be done with the unpack template `'b16'`. To obtain the individual bit values from the bit string we use `split` with the "empty" separator pattern which dissects into individual characters. Bit values from the "reserved" positions are simply assigned to `undef`, a convenient notation for "I don't care where this goes".

```
($carry, undef, $parity, undef, $auxcarry, undef, $zero, $sign,
 $trace, $interrupt, $direction, $overflow) =
   split( //, unpack( 'b16', $status ) );
```

We could have used an unpack template `'b12'` just as well, since the last 4 bits can be ignored anyway.

Uuencoding

Another odd-man-out in the template alphabet is u, which packs an "uuencoded string". ("uu" is short for Unix-to-Unix.) Chances are that you won't ever need this encoding technique which was invented to overcome the shortcomings of old-fashioned transmission mediums that do not support other than simple ASCII data. The essential recipe is simple: Take three bytes, or 24 bits. Split them into 4 six-packs, adding a space (0x20) to each. Repeat until all of the data is blended. Fold groups of 4 bytes into lines no longer than 60 and garnish them in front with the original byte count (incremented by 0x20) and a `"\n"` at the end. - The `pack` chef will prepare this for you, a la minute, when you select pack code u on the menu:

```
my $uubuf = pack( 'u', $bindat );
```

A repeat count after u sets the number of bytes to put into an uuencoded line, which is the maximum of 45 by default, but could be set to some (smaller) integer multiple of three. `unpack` simply ignores the repeat count.

Doing Sums

An even stranger template code is %<*number*>. First, because it's used as a prefix to some other template code. Second, because it cannot be used in `pack` at all, and third, in `unpack`, doesn't return the data as defined by the template code it precedes. Instead it'll give you an integer of *number* bits that is computed from the data value by doing sums. For numeric unpack codes, no big feat is

achieved:

```
my $buf = pack( 'iii', 100, 20, 3 );
print unpack( '%32i3', $buf ), "\n";  # prints 123
```

For string values, % returns the sum of the byte values saving you the trouble of a sum loop with `substr` and `ord`:

```
print unpack( '%32A*', "\x01\x10" ), "\n";  # prints 17
```

Although the % code is documented as returning a "checksum": don't put your trust in such values! Even when applied to a small number of bytes, they won't guarantee a noticeable Hamming distance.

In connection with b or B, % simply adds bits, and this can be put to good use to count set bits efficiently:

```
my $bitcount = unpack( '%32b*', $mask );
```

And an even parity bit can be determined like this:

```
my $evenparity = unpack( '%1b*', $mask );
```

Unicode

Unicode is a character set that can represent most characters in most of the world's languages, providing room for over one million different characters. Unicode 3.1 specifies 94,140 characters: The Basic Latin characters are assigned to the numbers 0 - 127. The Latin-1 Supplement with characters that are used in several European languages is in the next range, up to 255. After some more Latin extensions we find the character sets from languages using non-Roman alphabets, interspersed with a variety of symbol sets such as currency symbols, Zapf Dingbats or Braille. (You might want to visit *http://www.unicode.org/* for a look at some of them - my personal favourites are Telugu and Kannada.)

The Unicode character sets associates characters with integers. Encoding these numbers in an equal number of bytes would more than double the requirements for storing texts written in Latin alphabets. The UTF-8 encoding avoids this by storing the most common (from a western point of view) characters in a single byte while encoding the rarer ones in three or more bytes.

Perl uses UTF-8, internally, for most Unicode strings.

So what has this got to do with `pack`? Well, if you want to compose a Unicode string (that is internally encoded as UTF-8), you can do so by using template code U. As an example, let's produce the Euro currency symbol (code number 0x20AC):

```
$UTF8{Euro} = pack( 'U', 0x20AC );
# Equivalent to: $UTF8{Euro} = "\x{20ac}";
```

Inspecting `$UTF8{Euro}` shows that it contains 3 bytes: "\xe2\x82\xac". However, it contains only 1 character, number 0x20AC. The round trip can be completed with `unpack`:

```
$Unicode{Euro} = unpack( 'U', $UTF8{Euro} );
```

Unpacking using the U template code also works on UTF-8 encoded byte strings.

Usually you'll want to pack or unpack UTF-8 strings:

```
# pack and unpack the Hebrew alphabet
my $alefbet = pack( 'U*', 0x05d0..0x05ea );
my @hebrew = unpack( 'U*', $utf );
```

Please note: in the general case, you're better off using Encode::decode_utf8 to decode a UTF-8 encoded byte string to a Perl Unicode string, and Encode::encode_utf8 to encode a Perl Unicode string to UTF-8 bytes. These functions provide means of handling invalid byte sequences and generally have a friendlier interface.

Another Portable Binary Encoding

The pack code `w` has been added to support a portable binary data encoding scheme that goes way beyond simple integers. (Details can be found at *http://Casbah.org/*, the Scarab project.) A BER (Binary Encoded Representation) compressed unsigned integer stores base 128 digits, most significant digit first, with as few digits as possible. Bit eight (the high bit) is set on each byte except the last. There is no size limit to BER encoding, but Perl won't go to extremes.

```
my $berbuf = pack( 'w*', 1, 128, 128+1, 128*128+127 );
```

A hex dump of `$berbuf`, with spaces inserted at the right places, shows 01 8100 8101 81807F. Since the last byte is always less than 128, `unpack` knows where to stop.

Template Grouping

Prior to Perl 5.8, repetitions of templates had to be made by x-multiplication of template strings. Now there is a better way as we may use the pack codes `(` and `)` combined with a repeat count. The `unpack` template from the Stack Frame example can simply be written like this:

```
unpack( 'v2 (vXXCC)5 v5', $frame )
```

Let's explore this feature a little more. We'll begin with the equivalent of

```
join( '', map( substr( $_, 0, 1 ), @str ) )
```

which returns a string consisting of the first character from each string. Using pack, we can write

```
pack( '(A)'.@str, @str )
```

or, because a repeat count * means "repeat as often as required", simply

```
pack( '(A)*', @str )
```

(Note that the template `A*` would only have packed `$str[0]` in full length.)

To pack dates stored as triplets (day, month, year) in an array `@dates` into a sequence of byte, byte, short integer we can write

```
$pd = pack( '(CCS)*', map( @$_, @dates ) );
```

To swap pairs of characters in a string (with even length) one could use several techniques. First, let's use x and X to skip forward and back:

```
$s = pack( '(A)*', unpack( '(xAXXAx)*', $s ) );
```

We can also use @ to jump to an offset, with 0 being the position where we were when the last `(` was encountered:

```
$s = pack( '(A)*', unpack( '(@1A @0A @2)*', $s ) );
```

Finally, there is also an entirely different approach by unpacking big endian shorts and packing them in the reverse byte order:

```
$s = pack( '(v)*', unpack( '(n)*', $s );
```

Lengths and Widths

String Lengths

In the previous section we've seen a network message that was constructed by prefixing the binary message length to the actual message. You'll find that packing a length followed by so many bytes of data is a frequently used recipe since appending a null byte won't work if a null byte may be part of the data. Here is an example where both techniques are used: after two null terminated strings with source and destination address, a Short Message (to a mobile phone) is sent after a length byte:

```
my $msg = pack( 'Z*Z*CA*', $src, $dst, length( $sm ), $sm );
```

Unpacking this message can be done with the same template:

```
( $src, $dst, $len, $sm ) = unpack( 'Z*Z*CA*', $msg );
```

There's a subtle trap lurking in the offing: Adding another field after the Short Message (in variable `$sm`) is all right when packing, but this cannot be unpacked naively:

```
# pack a message
my $msg = pack( 'Z*Z*CA*C', $src, $dst, length( $sm ), $sm, $prio );

# unpack fails - $prio remains undefined!
( $src, $dst, $len, $sm, $prio ) = unpack( 'Z*Z*CA*C', $msg );
```

The pack code `A*` gobbles up all remaining bytes, and `$prio` remains undefined! Before we let disappointment dampen the morale: Perl's got the trump card to make this trick too, just a little further up the sleeve. Watch this:

```
# pack a message: ASCIIZ, ASCIIZ, length/string, byte
my $msg = pack( 'Z* Z* C/A* C', $src, $dst, $sm, $prio );

# unpack
( $src, $dst, $sm, $prio ) = unpack( 'Z* Z* C/A* C', $msg );
```

Combining two pack codes with a slash (`/`) associates them with a single value from the argument list. In `pack`, the length of the argument is taken and packed according to the first code while the argument itself is added after being converted with the template code after the slash. This saves us the trouble of inserting the `length` call, but it is in `unpack` where we really score: The value of the length byte marks the end of the string to be taken from the buffer. Since this combination doesn't make sense except when the second pack code isn't `a*`, `A*` or `Z*`, Perl won't let you.

The pack code preceding `/` may be anything that's fit to represent a number: All the numeric binary pack codes, and even text codes such as `A4` or `Z*`:

```
# pack/unpack a string preceded by its length in ASCII
my $buf = pack( 'A4/A*', "Humpty-Dumpty" );
# unpack $buf: '13  Humpty-Dumpty'
my $txt = unpack( 'A4/A*', $buf );
```

`/` is not implemented in Perls before 5.6, so if your code is required to work on older Perls you'll need to `unpack( 'Z* Z* C')` to get the length, then use it to make a new unpack string. For example

```
   # pack a message: ASCIIZ, ASCIIZ, length, string, byte (5.005
compatible)
   my $msg = pack( 'Z* Z* C A* C', $src, $dst, length $sm, $sm, $prio );
```

```
# unpack
( undef, undef, $len) = unpack( 'Z* Z* C', $msg );
($src, $dst, $sm, $prio) = unpack ( "Z* Z* x A$len C", $msg );
```

But that second `unpack` is rushing ahead. It isn't using a simple literal string for the template. So maybe we should introduce...

Dynamic Templates

So far, we've seen literals used as templates. If the list of pack items doesn't have fixed length, an expression constructing the template is required (whenever, for some reason, `()*` cannot be used). Here's an example: To store named string values in a way that can be conveniently parsed by a C program, we create a sequence of names and null terminated ASCII strings, with = between the name and the value, followed by an additional delimiting null byte. Here's how:

```
my $env = pack( '(A*A*Z*)' . keys( %Env ) . 'C',
                map( { ( $_, '=', $Env{$_} ) } keys( %Env ) ), 0 );
```

Let's examine the cogs of this byte mill, one by one. There's the `map` call, creating the items we intend to stuff into the `$env` buffer: to each key (in `$_`) it adds the = separator and the hash entry value. Each triplet is packed with the template code sequence `A*A*Z*` that is repeated according to the number of keys. (Yes, that's what the `keys` function returns in scalar context.) To get the very last null byte, we add a `0` at the end of the `pack` list, to be packed with `C`. (Attentive readers may have noticed that we could have omitted the 0.)

For the reverse operation, we'll have to determine the number of items in the buffer before we can let `unpack` rip it apart:

```
my $n = $env =~ tr/\0// - 1;
my %env = map( split( /=/, $_ ), unpack( "(Z*)$n", $env ) );
```

The `tr` counts the null bytes. The `unpack` call returns a list of name-value pairs each of which is taken apart in the `map` block.

Counting Repetitions

Rather than storing a sentinel at the end of a data item (or a list of items), we could precede the data with a count. Again, we pack keys and values of a hash, preceding each with an unsigned short length count, and up front we store the number of pairs:

```
my $env = pack( 'S(S/A* S/A*)*', scalar keys( %Env ), %Env );
```

This simplifies the reverse operation as the number of repetitions can be unpacked with the `/` code:

```
my %env = unpack( 'S/(S/A* S/A*)', $env );
```

Note that this is one of the rare cases where you cannot use the same template for `pack` and `unpack` because `pack` can't determine a repeat count for a `()`-group.

Intel HEX

Intel HEX is a file format for representing binary data, mostly for programming various chips, as a text file. (See *http://en.wikipedia.org/wiki/.hex* for a detailed description, and *http://en.wikipedia.org/wiki/SREC_(file_format)* for the Motorola S-record format, which can be unravelled using the same technique.) Each line begins with a colon (':') and is followed by a sequence of hexadecimal characters, specifying a byte count *n* (8 bit), an address (16 bit, big endian), a record type (8 bit), *n* data bytes and a checksum (8 bit) computed as the least significant byte of the two's complement sum of the preceding bytes. Example: `:0300300002337A1E`.

The first step of processing such a line is the conversion, to binary, of the hexadecimal data, to obtain the four fields, while checking the checksum. No surprise here: we'll start with a simple `pack` call to convert everything to binary:

```
my $binrec = pack( 'H*', substr( $hexrec, 1 ) );
```

The resulting byte sequence is most convenient for checking the checksum. Don't slow your program down with a for loop adding the `ord` values of this string's bytes - the `unpack` code `%` is the thing to use for computing the 8-bit sum of all bytes, which must be equal to zero:

```
die unless unpack( "%8C*", $binrec ) == 0;
```

Finally, let's get those four fields. By now, you shouldn't have any problems with the first three fields - but how can we use the byte count of the data in the first field as a length for the data field? Here the codes `x` and `X` come to the rescue, as they permit jumping back and forth in the string to unpack.

```
my( $addr, $type, $data ) = unpack( "x n C X4 C x3 /a", $bin );
```

Code `x` skips a byte, since we don't need the count yet. Code `n` takes care of the 16-bit big-endian integer address, and `C` unpacks the record type. Being at offset 4, where the data begins, we need the count. `X4` brings us back to square one, which is the byte at offset 0. Now we pick up the count, and zoom forth to offset 4, where we are now fully furnished to extract the exact number of data bytes, leaving the trailing checksum byte alone.

Packing and Unpacking C Structures

In previous sections we have seen how to pack numbers and character strings. If it were not for a couple of snags we could conclude this section right away with the terse remark that C structures don't contain anything else, and therefore you already know all there is to it. Sorry, no: read on, please.

If you have to deal with a lot of C structures, and don't want to hack all your template strings manually, you'll probably want to have a look at the CPAN module `Convert::Binary::C`. Not only can it parse your C source directly, but it also has built-in support for all the odds and ends described further on in this section.

The Alignment Pit

In the consideration of speed against memory requirements the balance has been tilted in favor of faster execution. This has influenced the way C compilers allocate memory for structures: On architectures where a 16-bit or 32-bit operand can be moved faster between places in memory, or to or from a CPU register, if it is aligned at an even or multiple-of-four or even at a multiple-of eight address, a C compiler will give you this speed benefit by stuffing extra bytes into structures. If you don't cross the C shoreline this is not likely to cause you any grief (although you should care when you design large data structures, or you want your code to be portable between architectures (you do want that, don't you?)).

To see how this affects `pack` and `unpack`, we'll compare these two C structures:

```
typedef struct {
  char     c1;
  short    s;
  char     c2;
  long     l;
} gappy_t;

typedef struct {
  long     l;
  short    s;
```

```
  char     c1;
  char     c2;
} dense_t;
```

Typically, a C compiler allocates 12 bytes to a `gappy_t` variable, but requires only 8 bytes for a `dense_t`. After investigating this further, we can draw memory maps, showing where the extra 4 bytes are hidden:

```
0           +4          +8          +12
+--+--+--+--+--+--+--+--+--+--+--+--+
|c1|xx|  s  |c2|xx|xx|xx|     l     |    xx = fill byte
+--+--+--+--+--+--+--+--+--+--+--+--+
gappy_t

0           +4          +8
+--+--+--+--+--+--+--+--+
|     l     |  h  |c1|c2|
+--+--+--+--+--+--+--+--+
dense_t
```

And that's where the first quirk strikes: `pack` and `unpack` templates have to be stuffed with `x` codes to get those extra fill bytes.

The natural question: "Why can't Perl compensate for the gaps?" warrants an answer. One good reason is that C compilers might provide (non-ANSI) extensions permitting all sorts of fancy control over the way structures are aligned, even at the level of an individual structure field. And, if this were not enough, there is an insidious thing called `union` where the amount of fill bytes cannot be derived from the alignment of the next item alone.

OK, so let's bite the bullet. Here's one way to get the alignment right by inserting template codes `x`, which don't take a corresponding item from the list:

```
my $gappy = pack( 'cxs cxxx l!', $c1, $s, $c2, $l );
```

Note the `!` after `l`: We want to make sure that we pack a long integer as it is compiled by our C compiler. And even now, it will only work for the platforms where the compiler aligns things as above. And somebody somewhere has a platform where it doesn't. [Probably a Cray, where `shorts`, `ints` and `longs` are all 8 bytes. :-)]

Counting bytes and watching alignments in lengthy structures is bound to be a drag. Isn't there a way we can create the template with a simple program? Here's a C program that does the trick:

```
#include <stdio.h>
#include <stddef.h>

typedef struct {
  char     fc1;
  short    fs;
  char     fc2;
  long     fl;
} gappy_t;

#define Pt(struct,field,tchar) \
  printf( "@%d%s ", offsetof(struct,field), # tchar );

int main() {
```

```
    Pt( gappy_t, fc1, c  );
    Pt( gappy_t, fs,  s! );
    Pt( gappy_t, fc2, c  );
    Pt( gappy_t, fl,  l! );
    printf( "\n" );
  }
```

The output line can be used as a template in a `pack` or `unpack` call:

```
my $gappy = pack( '@0c @2s! @4c @8l!', $c1, $s, $c2, $l );
```

Gee, yet another template code - as if we hadn't plenty. But `@` saves our day by enabling us to specify the offset from the beginning of the pack buffer to the next item: This is just the value the `offsetof` macro (defined in `<stddef.h>`) returns when given a `struct` type and one of its field names ("member-designator" in C standardese).

Neither using offsets nor adding `x`'s to bridge the gaps is satisfactory. (Just imagine what happens if the structure changes.) What we really need is a way of saying "skip as many bytes as required to the next multiple of N". In fluent Templatese, you say this with `x!N` where N is replaced by the appropriate value. Here's the next version of our struct packaging:

```
my $gappy = pack( 'c x!2 s c x!4 l!', $c1, $s, $c2, $l );
```

That's certainly better, but we still have to know how long all the integers are, and portability is far away. Rather than 2, for instance, we want to say "however long a short is". But this can be done by enclosing the appropriate pack code in brackets: `[s]`. So, here's the very best we can do:

```
my $gappy = pack( 'c x![s] s c x![l!] l!', $c1, $s, $c2, $l );
```

Dealing with Endian-ness

Now, imagine that we want to pack the data for a machine with a different byte-order. First, we'll have to figure out how big the data types on the target machine really are. Let's assume that the longs are 32 bits wide and the shorts are 16 bits wide. You can then rewrite the template as:

```
my $gappy = pack( 'c x![s] s c x![l] l', $c1, $s, $c2, $l );
```

If the target machine is little-endian, we could write:

```
my $gappy = pack( 'c x![s] s< c x![l] l<', $c1, $s, $c2, $l );
```

This forces the short and the long members to be little-endian, and is just fine if you don't have too many struct members. But we could also use the byte-order modifier on a group and write the following:

```
my $gappy = pack( '( c x![s] s c x![l] l )<', $c1, $s, $c2, $l );
```

This is not as short as before, but it makes it more obvious that we intend to have little-endian byte-order for a whole group, not only for individual template codes. It can also be more readable and easier to maintain.

Alignment, Take 2

I'm afraid that we're not quite through with the alignment catch yet. The hydra raises another ugly head when you pack arrays of structures:

```
typedef struct {
  short    count;
```

```
  char     glyph;
} cell_t;

typedef cell_t buffer_t[BUFLEN];
```

Where's the catch? Padding is neither required before the first field `count`, nor between this and the next field `glyph`, so why can't we simply pack like this:

```
# something goes wrong here:
pack( 's!a' x @buffer,
      map{ ( $_->{count}, $_->{glyph} ) } @buffer );
```

This packs `3*@buffer` bytes, but it turns out that the size of `buffer_t` is four times `BUFLEN`! The moral of the story is that the required alignment of a structure or array is propagated to the next higher level where we have to consider padding *at the end* of each component as well. Thus the correct template is:

```
pack( 's!ax' x @buffer,
      map{ ( $_->{count}, $_->{glyph} ) } @buffer );
```

Alignment, Take 3

And even if you take all the above into account, ANSI still lets this:

```
typedef struct {
  char     foo[2];
} foo_t;
```

vary in size. The alignment constraint of the structure can be greater than any of its elements. [And if you think that this doesn't affect anything common, dismember the next cellphone that you see. Many have ARM cores, and the ARM structure rules make `sizeof (foo_t)` == 4]

Pointers for How to Use Them

The title of this section indicates the second problem you may run into sooner or later when you pack C structures. If the function you intend to call expects a, say, `void *` value, you *cannot* simply take a reference to a Perl variable. (Although that value certainly is a memory address, it's not the address where the variable's contents are stored.)

Template code `P` promises to pack a "pointer to a fixed length string". Isn't this what we want? Let's try:

```
# allocate some storage and pack a pointer to it
my $memory = "\x00" x $size;
my $memptr = pack( 'P', $memory );
```

But wait: doesn't `pack` just return a sequence of bytes? How can we pass this string of bytes to some C code expecting a pointer which is, after all, nothing but a number? The answer is simple: We have to obtain the numeric address from the bytes returned by `pack`.

```
my $ptr = unpack( 'L!', $memptr );
```

Obviously this assumes that it is possible to typecast a pointer to an unsigned long and vice versa, which frequently works but should not be taken as a universal law. - Now that we have this pointer the next question is: How can we put it to good use? We need a call to some C function where a pointer is expected. The read(2) system call comes to mind:

```
ssize_t read(int fd, void *buf, size_t count);
```

After reading *perlfunc* explaining how to use `syscall` we can write this Perl function copying a file to standard output:

```
require 'syscall.ph';
sub cat($){
    my $path = shift();
    my $size = -s $path;
    my $memory = "\x00" x $size;  # allocate some memory
    my $ptr = unpack( 'L', pack( 'P', $memory ) );
    open( F, $path ) || die( "$path: cannot open ($!)\n" );
    my $fd = fileno(F);
    my $res = syscall( &SYS_read, fileno(F), $ptr, $size );
    print $memory;
    close( F );
}
```

This is neither a specimen of simplicity nor a paragon of portability but it illustrates the point: We are able to sneak behind the scenes and access Perl's otherwise well-guarded memory! (Important note: Perl's `syscall` does *not* require you to construct pointers in this roundabout way. You simply pass a string variable, and Perl forwards the address.)

How does `unpack` with `P` work? Imagine some pointer in the buffer about to be unpacked: If it isn't the null pointer (which will smartly produce the `undef` value) we have a start address - but then what? Perl has no way of knowing how long this "fixed length string" is, so it's up to you to specify the actual size as an explicit length after `P`.

```
my $mem = "abcdefghijklmn";
print unpack( 'P5', pack( 'P', $mem ) ); # prints "abcde"
```

As a consequence, `pack` ignores any number or `*` after `P`.

Now that we have seen `P` at work, we might as well give `p` a whirl. Why do we need a second template code for packing pointers at all? The answer lies behind the simple fact that an `unpack` with `p` promises a null-terminated string starting at the address taken from the buffer, and that implies a length for the data item to be returned:

```
my $buf = pack( 'p', "abc\x00efhijklmn" );
print unpack( 'p', $buf );    # prints "abc"
```

Albeit this is apt to be confusing: As a consequence of the length being implied by the string's length, a number after pack code `p` is a repeat count, not a length as after `P`.

Using `pack(..., $x)` with `P` or `p` to get the address where `$x` is actually stored must be used with circumspection. Perl's internal machinery considers the relation between a variable and that address as its very own private matter and doesn't really care that we have obtained a copy. Therefore:

- Do not use `pack` with `p` or `P` to obtain the address of variable that's bound to go out of scope (and thereby freeing its memory) before you are done with using the memory at that address.
- Be very careful with Perl operations that change the value of the variable. Appending something to the variable, for instance, might require reallocation of its storage, leaving you with a pointer into no-man's land.
- Don't think that you can get the address of a Perl variable when it is stored as an integer or double number! `pack('P', $x)` will force the variable's internal representation to string, just as if you had written something like `$x .= ''`.

It's safe, however, to P- or p-pack a string literal, because Perl simply allocates an anonymous

Pack Recipes

Here are a collection of (possibly) useful canned recipes for `pack` and `unpack`:

```
# Convert IP address for socket functions
pack( "C4", split /\./, "123.4.5.6" );

# Count the bits in a chunk of memory (e.g. a select vector)
unpack( '%32b*', $mask );

# Determine the endianness of your system
$is_little_endian = unpack( 'c', pack( 's', 1 ) );
$is_big_endian = unpack( 'xc', pack( 's', 1 ) );

# Determine the number of bits in a native integer
$bits = unpack( '%32I!', ~0 );

# Prepare argument for the nanosleep system call
my $timespec = pack( 'L!L!', $secs, $nanosecs );
```

For a simple memory dump we unpack some bytes into just as many pairs of hex digits, and use `map` to handle the traditional spacing - 16 bytes to a line:

```
my $i;
print map( ++$i % 16 ? "$_ " : "$_\n",
           unpack( 'H2' x length( $mem ), $mem ) ),
      length( $mem ) % 16 ? "\n" : '';
```

Funnies Section

```
# Pulling digits out of nowhere...
print unpack( 'C', pack( 'x' ) ),
      unpack( '%B*', pack( 'A' ) ),
      unpack( 'H', pack( 'A' ) ),
      unpack( 'A', unpack( 'C', pack( 'A' ) ) ), "\n";

# One for the road ;-)
my $advice = pack( 'all u can in a van' );
```

Authors

Simon Cozens and Wolfgang Laun.