### 70.2.4 Offsets

Perl provides the function `sv_chop` to efficiently remove characters from the beginning of a string; you give it an SV and a pointer to somewhere inside the PV, and it discards everything before the pointer. The efficiency comes by means of a little hack: instead of actually removing the characters, `sv_chop` sets the flag `OOK` (offset OK) to signal to other functions that the offset hack is in effect, and it puts the number of bytes chopped off into the IV field of the SV. It then moves the PV pointer (called `SvPVX`) forward that many bytes, and adjusts `SvCUR` and `SvLEN`.

Hence, at this point, the start of the buffer that we allocated lives at `SvPVX(sv) - SvIV(sv)` in memory and the PV pointer is pointing into the middle of this allocated storage.

This is best demonstrated by example:

```
% ./perl -Ilib -MDevel::Peek -le '$a="12345"; $a=~s/.//; Dump($a)'
SV = PVIV(0x8128450) at 0x81340f0
  REFCNT = 1
  FLAGS = (POK,OOK,pPOK)
  IV = 1  (OFFSET)
  PV = 0x8135781 ( "1" . ) "2345"\0
  CUR = 4
  LEN = 5
```

Here the number of bytes chopped off (1) is put into IV, and `Devel::Peek::Dump` helpfully reminds us that this is an offset. The portion of the string between the "real" and the "fake" beginnings is shown in parentheses, and the values of `SvCUR` and `SvLEN` reflect the fake beginning, not the real one.

Something similar to the offset hack is performed on AVs to enable efficient shifting and splicing off the beginning of the array; while `AvARRAY` points to the first element in the array that is visible from Perl, `AvALLOC` points to the real start of the C array. These are usually the same, but a `shift` operation can be carried out by increasing `AvARRAY` by one and decreasing `AvFILL` and `AvLEN`. Again, the location of the real start of the C array only comes into play when freeing the array. See `av_shift` in *av.c*.

### 70.2.5 What's Really Stored in an SV?

Recall that the usual method of determining the type of scalar you have is to use `Sv*OK` macros. Because a scalar can be both a number and a string, usually these macros will always return TRUE and calling the `Sv*V` macros will do the appropriate conversion of string to integer/double or integer/double to string.

If you *really* need to know if you have an integer, double, or string pointer in an SV, you can use the following three macros instead:

```
SvIOKp(SV*)
SvNOKp(SV*)
SvPOKp(SV*)
```

These will tell you if you truly have an integer, double, or string pointer stored in your SV. The "p" stands for private.

The are various ways in which the private and public flags may differ. For example, a tied SV may have a valid underlying value in the IV slot (so SvIOKp is true), but the data should be accessed via the FETCH routine rather than directly, so SvIOK is false. Another is when numeric conversion has occured and precision has been lost: only the private flag is set on 'lossy' values. So when an NV is converted to an IV with loss, SvIOKp, SvNOKp and SvNOK will be set, while SvIOK wont be.

In general, though, it's best to use the `Sv*V` macros.