A layer does not have to implement all the functions, but the whole table has to be present. Unimplemented slots can be NULL (which will result in an error when called) or can be filled in with stubs to "inherit" behaviour from a "base class". This "inheritance" is fixed for all instances of the layer, but as the layer chooses which stubs to populate the table, limited "multiple inheritance" is possible.

74.2.6 Per-instance Data

The per-instance data are held in memory beyond the basic PerlIOl struct, by making a PerlIOl the first member of the layer's struct thus:

```
typedef struct
{
 struct _PerlIO base;       /* Base "class" info */
 STDCHAR *      buf;        /* Start of buffer */
 STDCHAR *      end;        /* End of valid part of buffer */
 STDCHAR *      ptr;        /* Current position in buffer */
 Off_t          posn;       /* Offset of buf into the file */
 Size_t         bufsiz;     /* Real size of buffer */
 IV             oneword;    /* Emergency buffer */
} PerlIOBuf;
```

In this way (as for perl's scalars) a pointer to a PerlIOBuf can be treated as a pointer to a PerlIOl.

74.2.7 Layers in action.

```
            table           perlio         unix
        |           |
        +-----------+    +----------+    +--------+
PerlIO ->|           |--->|  next    |--->|  NULL  |
        +-----------+    +----------+    +--------+
        |           |    |  buffer  |    |   fd   |
        +-----------+    |          |    +--------+
        |           |    +----------+
```

The above attempts to show how the layer scheme works in a simple case. The application's `PerlIO *` points to an entry in the table(s) representing open (allocated) handles. For example the first three slots in the table correspond to `stdin`,`stdout` and `stderr`. The table in turn points to the current "top" layer for the handle - in this case an instance of the generic buffering layer "perlio". That layer in turn points to the next layer down - in this case the lowlevel "unix" layer.

The above is roughly equivalent to a "stdio" buffered stream, but with much more flexibility:

- If Unix level `read`/`write`/`lseek` is not appropriate for (say) sockets then the "unix" layer can be replaced (at open time or even dynamically) with a "socket" layer.
- Different handles can have different buffering schemes. The "top" layer could be the "mmap" layer if reading disk files was quicker using `mmap` than `read`. An "unbuffered" stream can be implemented simply by not having a buffer layer.
- Extra layers can be inserted to process the data as it flows through. This was the driving need for including the scheme in perl 5.7.0+ - we needed a mechanism to allow data to be translated between perl's internal encoding (conceptually at least Unicode as UTF-8), and the "native" format used by the system. This is provided by the ":encoding(xxxx)" layer which typically sits above the buffering layer.
- A layer can be added that does "\n" to CRLF translation. This layer can be used on any platform, not just those that normally do such things.