**Interpreter**

And of course, there's the core of the Perl interpreter itself. Let's have a look at that in a little more detail.

Before we leave looking at the layout, though, don't forget that *MANIFEST* contains not only the file names in the Perl distribution, but short descriptions of what's in them, too. For an overview of the important files, try this:

```
perl -lne 'print if /^[^\/]+\.[ch]\s+/' MANIFEST
```

### 76.1.8 Elements of the interpreter

The work of the interpreter has two main stages: compiling the code into the internal representation, or bytecode, and then executing it. Compiled code in *perlguts* explains exactly how the compilation stage happens.

Here is a short breakdown of perl's operation:

**Startup**

The action begins in *perlmain.c*. (or *miniperlmain.c* for miniperl) This is very high-level code, enough to fit on a single screen, and it resembles the code found in *perlembed*; most of the real action takes place in *perl.c*

First, *perlmain.c* allocates some memory and constructs a Perl interpreter:

```
1 PERL_SYS_INIT3(&argc,&argv,&env);
2
3 if (!PL_do_undump) {
4     my_perl = perl_alloc();
5     if (!my_perl)
6         exit(1);
7     perl_construct(my_perl);
8     PL_perl_destruct_level = 0;
9 }
```

Line 1 is a macro, and its definition is dependent on your operating system. Line 3 references `PL_do_undump`, a global variable - all global variables in Perl start with `PL_`. This tells you whether the current running program was created with the `-u` flag to perl and then *undump*, which means it's going to be false in any sane context.

Line 4 calls a function in *perl.c* to allocate memory for a Perl interpreter. It's quite a simple function, and the guts of it looks like this:

```
my_perl = (PerlInterpreter*)PerlMem_malloc(sizeof(PerlInterpreter));
```

Here you see an example of Perl's system abstraction, which we'll see later: `PerlMem_malloc` is either your system's `malloc`, or Perl's own `malloc` as defined in *malloc.c* if you selected that option at configure time.

Next, in line 7, we construct the interpreter; this sets up all the special variables that Perl needs, the stacks, and so on.

Now we pass Perl the command line options, and tell it to go:

```
exitstatus = perl_parse(my_perl, xs_init, argc, argv, (char **)NULL);
if (!exitstatus) {
    exitstatus = perl_run(my_perl);
}
```

`perl_parse` is actually a wrapper around `S_parse_body`, as defined in *perl.c*, which processes the command line options, sets up any statically linked XS modules, opens the program and calls `yyparse` to parse it.