



CHAPTER 11

Hacking Perl

- 11.1 The development process 304
- 11.2 Debugging aids 306
- 11.3 Creating a patch 317
- 11.4 Perl 6: the future of Perl 321
- 11.5 Further reading 323
- 11.6 Summary 323

Just like any other piece of software, Perl is not a finished product; it's still being developed and has a lively development community. Both the authors are regular contributors to Perl, and we'd like to encourage you to think about getting involved with Perl's continued maintenance and development. This chapter will tell you what you need to know to begin.

11.1 THE DEVELOPMENT PROCESS

Perl is developed in several “strands”—not least, the new development of Perl 6 (see section 11.4), which is occurring separately from the ongoing maintenance of Perl 5. Here we concentrate on the current development of Perl 5.

11.1.1 Perl versioning

Perl has two types of version number: versions before 5.6.0 used a number of the form `x.yyy_zz`; `x` was the major version number (Perl 4, Perl 5), `y` was the minor release number, and `z` was the patchlevel. Major releases represented, for instance, either a complete rewrite or a major upheaval of the internals; minor releases sometimes added non-essential functionality, and releases changing the patchlevel were

primarily to fix bugs. Releases where *z* was 50 or more were unstable developers' releases working toward the next minor release.

Since 5.6.0, Perl uses the more standard open source version numbering system—version numbers are of the form *x.y.z*; releases where *y* is even are stable releases, and releases where it is odd are part of the *development track*.

11.1.2 The development tracks

Perl development has four major aims: extending portability, fixing bugs, adding optimizations, and creating new language features. Patches to Perl are usually made against the latest copy of the development release; the very latest copy, stored in the Perl repository (see section 11.1.5), is usually called the *bleadperl*.

The *bleadperl* eventually becomes the new minor release, but patches are also picked up by the maintainer of the stable release for inclusion. There are no hard and fast rules, and everything is left to the discretion of the maintainer, but in general, patches that are bug fixes or that address portability concerns (which include taking advantage of new features in some platforms, such as large file support or 64-bit integers) are merged into the stable release as well, whereas new language features tend to be left until the next minor release. Optimizations may or may not be included, depending on their impact on the source.

11.1.3 The perl5-porters mailing list

All Perl development happens on the *perl5-porters* (P5P) mailing list; if you plan to get involved, a subscription to this list is essential.

You can subscribe by sending an email to perl5-porters-subscribe@perl.org; you'll be asked to send an email to confirm, and then you should begin receiving mail from the list. To send mail to the list, address the mail to perl5-porters@perl.org; you don't have to be subscribed to post, and the list is not moderated. If, for whatever reason, you decide to unsubscribe, simply mail perl5-porters-unsubscribe@perl.org.

The list usually receives between 200 and 400 emails per week. If this is too much mail for you, you can subscribe instead to a daily digest service by emailing perl5-porters-digest-subscribe@perl.org.

There is also a *perl5-porters* FAQ (<http://simon-cozens.org/writings/p5p.faq>) that explains a lot of this information, plus more about how to behave on P5P and how to submit patches to Perl.

11.1.4 Pumpkins and pumpkings

Development is very loosely organized around the release managers of the stable and development tracks; these are the two *pumpkins*.

Perl development can also be divided into several smaller subsystems: the regular expression engine, the configuration process, the documentation, and so on. Responsibility for each of these areas is known as a *pumpkin*, and hence those who semiofficially take responsibility for them are called pumpkings.

You're probably wondering about the silly names. They stem from the days before Perl was kept under version control; to avoid conflicts, people had to manually check out a chunk of the Perl source by announcing their intentions to the mailing list. While the list was discussing what this process should be called, one of Chip Salzenburg's co-workers told him about a system they used for preventing two people from using a tape drive at once: there was a stuffed pumpkin in the office, and nobody could use the drive unless they had the pumpkin.

11.1.5 The Perl repository

Now Perl is kept in a version control system called Perforce (<http://www.perforce.com/>), which is hosted by ActiveState, Inc. There is no public access to the system, but various methods have been devised to allow developers near-realtime access:

- *Archive of Perl Changes*—This FTP site (<ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/>) contains both the current state of all the maintained Perl versions and a directory of changes made to the repository.
- *rsync*—Because it's a little inconvenient to keep up to date using FTP, the directories are also available via the software synchronization protocol `rsync` (<http://rsync.samba.org/>). If you have `rsync` installed, you can synchronize your working directory with `bleadperl` by issuing the command

```
% rsync -avz rsync://ftp.linux.activestate.com/perl-current/
```

If you use this route, you should periodically add the `--delete` option to `rsync` to clean out any files that have been deleted from the repository. Once, a proposed feature and its test were both removed from Perl, and those following `bleadperl` by `rsync` reported test failures for a test that no longer existed.

- *Periodic snapshots*—The development pumpking releases periodic snapshots of `bleadperl`, particularly when an important change happens. These are usually available from a variety of URLs, and always from <ftp://ftp.funet.fi/pub/languages/perl/snap/>.

11.2 DEBUGGING AIDS

A number of tools are available to developers to help you find and examine bugs in Perl; these tools are, of course, also useful if you're creating XS extensions and applications with embedded Perl. There are four major categories:

- Perl modules such as `Devel::Peek`, which allow you to get information about Perl's operation
- `perl`'s own debugging mode
- Convenience functions built into `perl` that you can call to get debugging information
- External applications

11.2.1 Debugging modules

You saw in chapter 4 how the `Devel::Peek` module can dump information about SVs; you've also learned about the `B::Terse` module for dumping the op tree. The op tree diagrams in chapter 10 were produced using the CPAN module `B::Tree`. You can use other modules to get similar information.

Compiler modules

Due to the way the compiler works, you can use it to get a lot of information about the op tree. The most extensive information can be found using the `B::Debug` module, which dumps all the fields of all OPs and SVs in the op tree.

Another useful module is `B::Graph`, which produces the same information as `B::Debug` but does so in the form of a graph.

Other modules

The core module `re` has a debugging mode, use `re 'debug'`; which traces the execution of regular expressions. You can use it, for instance, to examine the regular expression engine's backtracking behavior:

```
% perl -e 'use re "debug"; "aaa" =~/\w+d/;'
Compiling REX '\w+d'
size 4 first at 2
  1: PLUS(3)
  2:  ALNUM(0)
  3: DIGIT(4)
  4: END(0)
stclass `ALNUM' plus minlen 2
Matching REX '\w+d' against `aaa'
Setting an EVAL scope, savestack=3
  0 <> <aaa>          | 1:  PLUS
                        ALNUM can match 3 times out of 32767...
Setting an EVAL scope, savestack=3
  3 <aaa> <>          | 3:  DIGIT
                        failed...
  2 <aa> <a>          | 3:  DIGIT
                        failed...
  1 <a> <aa>          | 3:  DIGIT
                        failed...
                        failed...
Freeing REX: '\w+d'
```

Turning to CPAN, you can use the `Devel::Leak` module to detect and trace memory leaks in perl, and `Devel::Symdump` is useful for dumping and examining the symbol table.

11.2.2 The built-in debugger: perl -D

If you configure Perl passing the flag `-Doptimize='-g'` to `Configure`, it will do two things: it will tell the C compiler to add special debugging information to the

object files it produces (you'll see how that's used in a moment), and it will define the preprocessor macro `DEBUGGING`, which turns on some special debugging options.

NOTE If you're running `Configure` interactively, you can turn on debugging as follows.

By default, Perl 5 compiles with the `-O` flag to use the optimizer. Alternately, you might want to use the symbolic debugger, which uses the `-g` flag (on traditional Unix systems). Either flag can be specified here. To use neither flag, specify the word `none`.

You should use the optimizer/debugger flag `[-O2] -g`.

Compiling `perl` like this allows you to use the `-D` flag on the `perl` command line to select the level of debugging you require. The most useful debugging options are as follows (see the `perlrun` documentation for a full list).

The `-Ds` option

This option turns on stack snapshots, printing a summary of what's on the argument stack each time an operation is performed. It is not too useful on its own, but is highly recommended when combined with the `-Dt` switch. Here you can see how Perl builds up lists by putting successive values onto the stack, and performs array assignments:

```
% perl -Ds -e '@a = (1,2,3)'
```

```
EXECUTING...
```

```
=>
=>
=>
=> *
=> * IV(1) ①
=> * IV(1) IV(2)
=> * IV(1) IV(2) IV(3)
=> * IV(1) IV(2) IV(3) * ②
=> * IV(1) IV(2) IV(3) * GV()
=> * IV(1) IV(2) IV(3) * AV()
=> ③
```

The array is first placed on the stack as a glob—an entry into the symbol table

The `rv2av` operator resolves the glob into an AV

- ① Perl pushes each of the values of the list onto the argument stack. The asterisk before the list represents an entry in the mark stack.
- ② Once the list has been built up, Perl places another mark between the right side of an assignment and the left side, so it knows how many elements are due for assignment.
- ③ Once the assignment has been made, everything from the first mark is popped off the stack.

The -Dt option

This option traces each individual op as it is executed. Let's see the previous code again, but this time with a listing of the ops:

```
% perl -Dst -e '@a = (1,2,3)'  
EXECUTING...  
  
=>  
(-e:0) enter  
=>  
(-e:0) nextstate  
=>  
(-e:1) pushmark  
=> *  
(-e:1) const(IV(1))  
=> * IV(1)  
(-e:1) const(IV(2))  
=> * IV(1) IV(2)  
(-e:1) const(IV(3))  
=> * IV(1) IV(2) IV(3)  
(-e:1) pushmark  
=> * IV(1) IV(2) IV(3) *  
(-e:1) gv(main::a)  
=> * IV(1) IV(2) IV(3) * GV()  
(-e:1) rv2av  
=> * IV(1) IV(2) IV(3) * AV()  
(-e:1) aassign  
=>  
(-e:1) leave
```

The -Dr option

The -Dr flag is identical to the use re 'debug'; module discussed earlier.

The -DI option

This option reports when perl reaches an ENTER or LEAVE statement, and reports on which line and in which file the statement occurred.

The -Dx option

This option is roughly equivalent to B::Terse. It produces a dump of the op tree using the op_dump function described later. It's a handy compromise between B::Terse and B::Debug.

The -Do option

This option turns on reporting of method resolution—that is, what happens when Perl calls a method on an object or class. For instance, it tells you when DESTROY methods are called, as well as what happens during inheritance lookups.

11.2.3 Debugging functions

The Perl core defines a number of functions to aid in debugging its internal goings-on. You can call them either from debugging sections of your own C or XS code or from a source-level debugger.

The sv_dump function

```
void sv_dump(SV* sv);
```

This function is roughly equivalent to the `Devel::Peek` module—it allows you to inspect any of Perl’s data types. The principle differences between this function and `Devel::Peek` is that it is not recursive—for instance, a reference will be dumped like this

```
SV = RV(0x814fd10) at 0x814ec80
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x814ec5c
```

and its referent is not automatically dumped. However, it does let you get at values that are not attached to a variable, such as arrays and scalars used to hold data internal to perl.

The op_dump function

```
void op_dump(OP* op);
```

The `-Dx` debugging option is implemented, essentially, by calling `op_dump(PL_mainroot)`. It takes an `op`; lists the `op`’s type, flags, and important additional fields; and recursively calls itself on the `op`’s children.

The dump_sub function

```
void dump_sub(GV* gv);
```

This function extracts the CV from a glob and runs `op_dump` on the root of its op tree.

11.2.4 External debuggers

There’s another way to debug your code, which is often more useful when you’re fiddling around in C. A *source level debugger* allows you to step through your C code line by line or function by function and execute C code on the fly, just as you’d do with the built-in Perl debugger.

Source-level debuggers come in many shapes and sizes: if you’re working in a graphical environment such as Microsoft Visual Studio, a debugging mode may be built into it. Just as with compilers, there are also command-line versions. In this section we’ll look at another free tool, the GNU Debugger (`gdb`); much of what we say is applicable to other similar debuggers, such as Solaris’s `dbx`.

Compiling for debugging

Unfortunately, before you can use the debugger on a C program, you must compile it with special options. As you've seen, the debugging option (usually `-g` on command-line compilers) embeds information into the binary detailing the file name and line number for each operation, so that the debugger can, for instance, stop at a specific line in a C source file.

So, before using the debugger, you must recompile Perl with the `-Doptimize='-g'` option to Configure, as shown in section 11.2.2.

Invoking the debugger

We'll assume you're using `gdb` and you've compiled Perl with the `-g` flag. If we type `gdb perl` in the directory in which you built Perl, we see the following:

```
% gdb perl
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb)
```

If, however, you see the words “(no debugging symbols found)”, you're either in the wrong place or you didn't compile Perl with debugging support.

You can type `help` at any time to get a summary of the commands, or type `quit` (or just press `Ctrl-D`) to leave the debugger.

You can run `perl` without any intervention from the debugger by simply typing `run`; doing so is equivalent to executing `perl` with no command-line options and means it will take a program from standard input.

To pass command-line options to `perl`, put them after the `run` command, like this:

```
(gdb) run -Ilib -MDevel::Peek -e '$a="X"; $a++; Dump($a)'
Starting program: /home/simon/patchbay/perl/perl -Ilib -MDevel::Peek
-e '$a="X"; $a++; Dump($a)'
SV = PV(0x8146fdc) at 0x8150a18
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x8154620 "Y"\0
  CUR = 1
  LEN = 2

Program exited normally
```

Setting breakpoints

Running through a program normally isn't very exciting. The most important thing to do is choose a place to freeze execution of the program, so you can examine further what's going on at that point.

The `break` command sets a *breakpoint*—a point in the program at which the debugger will halt execution and bring you back to the (`gdb`) prompt. You can give `break` either the name of a function or a location in the source code of the form `filename.c:lineno`. For instance, in the version of Perl installed here,¹ the main `op` dispatch code is at `run.c:53`:

```
(gdb) break run.c:53
Breakpoint 1 at 0x80ba331: file run.c, line 53.
```

This code sets breakpoint number 1, which will be triggered when execution gets to line 53 of `run.c`.

NOTE *Setting breakpoints*—Blank lines, or lines containing comments or preprocessor directives, will never be executed; but if you set a breakpoint on them, the debugger should stop at the next line containing code. This also applies to sections of code that are `#ifdef`'d out.

If you give `break` a function name, be sure to give the name in the `Perl_` namespace: that is, `Perl_runops_debug` instead of `runops_debug`.

When you use `run`, execution will halt when it gets to the specified place. `gdb` will display the number of the breakpoint that was triggered and the line of code in question:

```
(gdb) run -e1
Starting program: /home/simon/patchbay/perl/perl -e1

Breakpoint 1, Perl_runops_debug () at run.c:53
53          } while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX));
```

You can now use the `backtrace` command, `bt`, to examine the call stack and find out how you got there (`where` is also available as a synonym for `bt`):

```
(gdb) bt
#0 Perl_runops_debug () at run.c:53
#1 0x805dc9f in S_run_body (oldscope=1) at perl.c:1458
#2 0x805d871 in perl_run (my_perl=0x8146b98) at perl.c:1380
#3 0x805a4d5 in main (argc=2, argv=0xbffff8cc, env=0xbffff8d8)
  at perlmain.c:52
#4 0x40076dcc in __libc_start_main () from /lib/libc.so.6
```

This result tells us that we're currently in `Perl_runops_debug`, after being called by `S_run_body` on line 1380 of `perl.c`. `gdb` also displays the value of the

¹ 5.6.0. Don't worry if you get slightly different line numbers in your version.

arguments to each function, although many of them (those given as hexadecimal numbers) are pointers.

You can restart execution by typing `continue`; if the code containing a breakpoint is executed again, the debugger will halt once more. If not, the program will run until termination.

You can set multiple breakpoints simply by issuing more `break` commands. If multiple breakpoints are set, the debugger will stop each time execution reaches any of the breakpoints in force.

Unwanted breakpoints can be deleted using the `delete` command; on its own, `delete` will delete all breakpoints. To delete a given breakpoint, use `delete n`, where `n` is the number of the breakpoint.

To temporarily turn off a breakpoint, use the `disable` and `enable` commands.

Good breakpoints to choose when debugging `perl` include the `main` `op` dispatch code shown earlier, `main`, `S_parse_body`, `perl_construct`, `perl_destruct`, and `Perl_yyparse` (not for the faint of heart, because it places you right in the middle of the Yacc parser).

Stepping through a program

Although it's possible to work out the flow of execution just by using breakpoints, it's a lot easier to watch the statements as they are executed. The key commands to do this are `step`, `next`, and `finish`.

The `step` command traces the flow of execution step by step. Let's see what happens when we break at the `main` `op` dispatch loop and step through execution:

```
(gdb) run -e1
Starting program: /home/simon/patchbay/perl/perl -e1

Breakpoint 1, Perl_runops_debug () at run.c:53
53          } while ((PL_op = CALL_FPTR(PL_op->op_ppaddr) (aTHX)));
(gdb) step

Perl_pp_enter () at pp_hot.c:1587
1587         djSP;
(gdb) step
1589         I32 gimme = OP_GIMME(PL_op, -1);
(gdb)
1591         if (gimme == -1) {
(gdb)
1592             if (cxstack_ix >= 0)
(gdb)
1595                 gimme = G_SCALAR;
```

TIP Pressing Return repeats the last command.

As we stepped into the first `op`, `enter`, `gdb` loaded up `pp_hot.c` and entered the `Perl_pp_enter` function. The function in question begins like this:

```

1585 PP(pp_enter)
1586 {
1587     djSP;
1588     register PERL_CONTEXT *cx;
1589     I32 gimme = OP_GIMME(PL_op, -1);
1590
1591     if (gimme == -1) {
1592         if (cxstack_ix >= 0)
1593             gimme = cxstack[cxstack_ix].blk_gimme;
1594         else
1595             gimme = G_SCALAR;
1596     }
1597     ...

```

`gdb` first stopped at line 1587, which is the first line in the function. The first three lines of the function are, as you might expect, variable definitions. `gdb` does not normally stop on variable definitions unless they are also assignments. `djSP` happens to be a macro that expands to

```
register SV **sp = PL_stack_sp
```

declaring a local copy of the stack pointer. The next line, however, is not an assignment, which is why `step` causes `gdb` to move on to line 1589. `gdb` also skips blank space, so the next line it stops on is 1591.

Because the program enters the `if` statement, we know the `gimme` (the context in which this piece of Perl is being executed) is `-1`, signifying “not yet known.” Next we go from the inner `if` statement to the `else` branch, meaning that `cx_stack_ix`, the index into the context stack, is less than zero. Hence `gimme` is set to `G_SCALAR`.

In Perl terms, this means the context stack holds the context for each block; when you call a sub in list context, an entry is popped onto the context stack signifying this event. This entry allows the code that implements `return` to determine which context is expected. Because we are in the outermost block of the program, there are no entries on the context stack at the moment. The code we have just executed sets the context of the outer block to scalar context. (Unfortunately, `wantarray` is useful only inside a subroutine, so the usual way of demonstrating the context won’t work. You’ll have to take our word for it.)

Sometimes `step` is too slow, and you don’t want to descend into a certain function and execute every line in it. For instance, you’ll notice after a while that `ENTER` and `SAVETMPS` often appear next to each other and cause `Perl_push_scope` and `Perl_save_int` to be executed. If you’re not interested in debugging those functions, you can skip them using the `next` command. They will still be executed, but the debugger will not trace their execution:

```

Breakpoint 2, Perl_pp_enter () at pp_hot.c:1598
1598         ENTER;
(gdb) next
1600         SAVETMPS;
(gdb)
1601         PUSHBLOCK(cx, Cxt_BLOCK, SP);
(gdb)
1603         RETURN;
(gdb)

```

Alternatively, you can run the current function to its conclusion without tracing it by using the `finish` command:

```

(gdb) step
Perl_runops_debug () at run.c:42
42         PERL_ASYNC_CHECK();
(gdb)
43         if (PL_debug) {
(gdb)
53         } while ((PL_op = CALL_FPTR(PL_op->op_ppaddr) (aTHX)));
(gdb)
Perl_pp_nextstate () at pp_hot.c:37
37         PL_curcop = (COP*)PL_op;
(gdb) finish
Run till exit from #0 Perl_pp_nextstate () at pp_hot.c:37
0x80ba64b in Perl_runops_debug () at run.c:53
53         } while ((PL_op = CALL_FPTR(PL_op->op_ppaddr) (aTHX)));
Value returned is $1 = (OP *) 0x814cb68

```

Here we step over the main `op` dispatch loop until `Perl_pp_nextstate` is called. Because we're not particularly interested in that function, we call `finish` to let it run. The debugger then confirms that it's running `Perl_pp_nextstate` until the function exits and displays where it has returned to and the value returned from the function.

TIP *Emacs makes it easy*—If you're a user of the Emacs editor, you might find `gdb` major mode to be extremely helpful; it automatically opens any source files `gdb` refers to and can trace the flow of control in the source buffers. Thus it's easy for you to see what's going on around the source that's currently being executed.

Alternatives to gdb—If you're not a fan of command-line debugging, you may wish to investigate alternatives to `gdb`. For Windows users, Microsoft Visual C can't be beaten; for Unix users, Tim recommends `ddd` (Data Display Debugger), which is a graphical front-end to `gdb`. `ddd` extends the usual source-navigation functions of a debugger with an interactive graphical display of data, including arrays and structures.

Evaluating expressions

You can now perform most of the debugging you need with ease, but one more feature of `gdb` makes it even easier. The `print` command allows you to execute C expressions on the fly and display their results.

Unfortunately, there is one drawback: `gdb` doesn't know about preprocessor macros, so you must expand the macros yourself. For instance, to find the reference count of an `SV`, we can't say

```
(gdb) print SvREFCNT(sv)
No symbol "SvREFCNT" in current context.
```

Instead, we have to say

```
(gdb) print sv->sv_refcnt
$1=1
```

Or, to look at the contents of the `SV`,

```
(gdb) print *sv
$2 = {sv_any = 0x8147a10, sv_refcnt = 1, sv_flags = 536870923}
```

You can also use `print` to call C functions, such as the debugging functions mentioned earlier:

```
(gdb) print Perl_sv_dump(sv)
SV = PV(0x8146d14) at 0x8150824
  REFcnt = 1
  FLAGS = (POK,READONLY,pPOK)
  PV = 0x8151968 "hello"\0
  CUR = 5
  LEN = 6
$9 = void
```

Using these functions in conjunction with the execution-tracing commands of `gdb` should allow you to examine almost every area of Perl's internals.

Debugging XS code

There are a couple of little wrinkles when it comes to debugging XS modules. With XS, modules are usually dynamically loaded into memory; thus when `perl` starts, the functions aren't loaded—and when `gdb` starts, it can't find them.

The solution is to choose a breakpoint after the XS module has been dynamically loaded. A good place is `S_run_body`—here the `BEGIN` blocks have been processed and hence all use'd modules have been loaded. This is just before the main part of the script is executed. If this is too late for your debugging, another good place to stop is inside the dynamic loading module, `DynaLoader`. `XS_DynaLoader_dl_load_file` is called for each module that needs to be dynamically loaded.

NOTE Don't forget that to effectively debug an XS module, you must recompile it with the debugging flag, `-g`. The official way to do this is to run `Makefile.PL` as follows:

```
% perl Makefile.PL OPTIMIZE=-g
```

However, it's also possible to hack the `OPTIMIZE=` line in the `Makefile` itself (but don't tell anyone we said that).

The next small problem is that the names of XS functions are mangled from the names you give them in the `.xs` file. You should look at the `.c` file produced by `xsubpp` to determine the real function names. For instance, the XS function `sdbm_TIEHASH` in the XS code for the `SDBM_File` becomes `XS_SDBM_File_TIEHASH`.

The rules for this mangling are regular (section 6.11):

- 1 The `PREFIX` given in the XS file is removed from the function name. Hence, `sdbm_` is stripped off to leave `TIEHASH`.
- 2 The `PACKAGE` name (`SDBM_File`) undergoes “C-ification” (any package separators, `::`, are converted to underscores) and is added to the beginning of the name: `SDBM_File_TIEHASH`.
- 3 `XS_` is prefixed to the name to give `XS_SDBM_File_TIEHASH`.

11.3 CREATING A PATCH

Suppose you've noticed a problem and debugged it. Now what? If possible, you should fix it; then, if you want fame and immortality, you should submit that patch back to `perl5-porters`. Let's explore this process.

11.3.1 How to solve problems

You should keep in mind a few standard design goals when you're considering how to approach a Perl patch; quite a lot of unwritten folklore explains why certain patches feel better than others. Here is an incomplete list of some of the more important principles we've picked up over the years:

- The most important rule is that you may not break old code. Perl 5 can happily run some ancient code, even dating back to Perl 1 days; we pride ourselves on backward compatibility. Hence, nothing you do should break that compatibility. This rule has a few direct implications: adding new syntax is tricky. Adding new operators is basically impossible; if you wanted to introduce a `chip` operator that took a character off the beginning of a string, it would break any code that defined a `chip` subroutine itself.
- Solve problems as generally as possible. Platform-specific `ifdefs` are frowned upon unless absolutely and obviously necessary. Try to avoid repetition of code. If you have a good, general routine that can be used in other places of the Perl

core, move it out to a separate function and change the rest of the core to use it. For instance, we needed a way for Perl to perform arbitrary transformations on incoming data—for example, to mark it as UTF-8 encoded, or convert it between different character encodings. The initial idea was to extend the source filter mechanism to apply not just to the source file input, but also to any file-handle. However, the more general solution was an extension of the Perl I/O abstraction to a layered model where transformation functions could be applied to various layers; then source filters could be re-implemented in terms of this new I/O system.

- Change as *little* as possible to get the job done, especially when you're not well known as a solid porter. Sweeping changes scare people, whether or not they're correct. It's a lot easier to check a 10-line patch for potential bugs than a 100-line patch.
- Don't do it in the core unless it needs to be done in the core. If you can do it in a Perl module or an XS module, it's unlikely that you need to do it in the core. As an example, DBM capability was moved out of the core into a bunch of XS modules; this approach also had the advantage that you could switch between different DBM libraries at runtime, and you had the extensibility of the `tie` system that could be used for things other than DBMs.
- Try to avoid introducing restrictions, even on things you haven't thought of yet. Always leave the door open for more interesting work along the same lines. A good example is `lvalue` subroutines, which were introduced in Perl 5.6.0. Once you have `lvalue` subroutines, why not `lvalue` method calls or even `lvalue` overloaded operators?

Some of the goals are just ideas you have to pick up in time. They may depend on the outlook of the pumpking and any major work going on at the time. For instance, during the reorganization of the I/O system mentioned earlier, any file-handling patches were carefully scrutinized to make sure they wouldn't have to be rewritten once the new system was in place. Hence, it's not really possible to give hard-and-fast design goals; but if you stick to the list we've just provided, you won't go far wrong.

11.3.2 Autogenerated files

A number of files should not be patched directly, because they are generated from other (usually Perl) programs. Most of these files are clearly marked, but the most important of these deserves a special note: if you add a new function to the core, you *must* add an entry to the table at the end of `embed.pl`. Doing so ensures that a correct function prototype is generated and placed in `protos.h`, that any documentation for that function is automatically extracted, and that the namespace for the function is automatically handled. (See the following note.) The syntax for entries in the table is explained in the documentation file `perlguts.pod`.

NOTE Perl's internal functions are carefully named so that when Perl is embedded in another C program, they do not override any functions the C program defines. Hence, all internal functions should be named `Perl_something` (apart from static functions, which are by convention named `S_something`). `embed.h` uses a complicated system of automatically generated `#defines` to allow you to call your function as `something()` inside the core and in XSUBs, but `Perl_something` must be used by embedders.

You must remember to rerun `embed.pl` after adding this entry. The Make target `regen_headers` will call all the Perl programs that generate other files.

A special exception is `perly.c`, which is generated by running `byacc` on `perly.y` and then being fixed with a patch. In the *extraordinarily* unlikely event that you need to fiddle with the Perl grammar in `perly.y`, you can run the Make target `run_byacc` to call `byacc` and then fix the resulting C file; if you are changing `perly.y`, it's polite to drop the VMS porters mailing list (`vmperl@perl.org`) a copy of the patch, because they use a different process to generate `perly.c`.

For changes that involve autogenerated files, such as adding a function to the core or changing a function's prototype, you only need to provide a patch for the generating program and leave a note to the effect that `regen_headers` should be run. You should not include, for instance, a patch to `protos.h`.

11.3.3 The patch itself

Patching styles vary, but the recommended style for Perl is a unified `diff`. If you're changing a small number of files, copy, say, `sv.c` to `sv.c~`, make your changes, and then run

```
% diff -u sv.c~ sv.c > /tmp/patch
% diff -u sv.h~ sv.h >> /tmp/patch
```

and so on for each file you change.

If you are doing this, remember to run `diff` from the root of the Perl source directory. Hence, if we're patching XS files in `ext/`, we say

```
% diff -u ext/Devel/Peek/Peek.xs~ ext/Devel/Peek/Peek.xs
>> /tmp/patch
```

For larger patches, you may find it easier to do something like this:

```
/home/me/work % rsync -avz
rsync://ftp.linux.activestate.com/perl-current/ bleedperl
/home/me/work % cp -R bleedperl myperl
/home/me/work % cd myperl
/home/me/work/myperl % Make your changes...
/home/me/work/myperl % cd ..
/home/me/work % diff -ruN bleedperl myperl > /tmp/patch
```

This code will create a patch that turns the current `bleadperl` into your personal Perl source tree. If you do this, please remember to prune your patch for autogenerated files and also items that do not belong in the source distribution (any test data you have used, or messages about binary files).

NOTE *Makepatch*—An alternative tool that can make patching easier is Johan Vromans’ `makepatch`, available from `$CPAN/authors/id/JV/`. It automates many of the steps we’ve described. Some swear by it, but some of us are stuck in our ways and do things the old way...

11.3.4 Documentation

If you change a feature of Perl that is visible to the user, you must, must, must update the documentation. Patches are not complete if they do not contain documentation.

Remember that if you introduce a new warning or error, you need to document it in `pod/perldiag.pod`.

Perl 5.6.0 introduced a system for providing documentation for internal functions, similar to Java’s `javadoc`. This `apidoc` is extracted by `embed.pl` and ends up in two files: `pod/perlapi.pod` contains documentation for functions that are deemed suitable for XS authors,² and `pod/perlintern.pod` contains the documentation for all other functions (internal functions).

`apidoc` is simply POD embedded in C comments; you should be able to pick up how it is used by looking around the various C files. If you add `apidoc` to a function, you should turn on the `d` flag in that function’s `embed.pl` entry.

11.3.5 Testing

The `t/` directory in the Perl source tree contains many (294, at last count) regression test scripts that ensure Perl is behaving as it should. When you change something, you should make sure your changes have not caused any of the scripts to break—they have been specially designed to try as many unexpected interactions as possible.

You should also add tests to the suite if you change a feature, so that your changes aren’t disturbed by future patching activity. Tests are in the ordinary style used for modules, so remember to update the `1 . . n` line at the top of the test.

11.3.6 Submitting your patch

Once you’ve put together a patch that includes documentation and new tests, it’s time to submit it to P5P. Your subject line should include the tag `[PATCH]`, with optionally a version number or name, or the name of the file you’re patching: for example, `[PATCH bleadperl]` or `[PATCH sv.c]`. This line lets the pumpking easily distinguish possible patches to be integrated from the usual list discussion. You

² Chapter 5 of this book was developed by starting from `pod/perlapi.pod`, and, in fact, we contributed back some pieces of chapter 5 as `apidoc`.

should also put a brief description of what you're solving on the subject line: for instance, [PATCH bleed] Fix B::Terse indentation.

The body of your email should be a brief discussion of the problem (some Perl code that demonstrates the problem is adequate) and how you've solved it. Then insert your patch directly into the body of the email—try to avoid sending it as an attachment. Also, be careful with cutting-and-pasting your patch in, because doing so may corrupt line wrapping or convert tabs to spaces.

Once you're ready, take a deep breath and hit Send!

11.4 PERL 6: THE FUTURE OF PERL

While we were busily preparing this book, something significant happened—Perl 6 was announced. Let's look at what led up to this announcement, and where the Perl 6 effort has gotten since then.

11.4.1 A history

At the Perl Conference in July 2000, Chip Salzenburg called a brainstorming session meeting of some eminent members of the Perl community to discuss the state of Perl. Chip wanted some form of “Perl Constitution” to resolve perceived problems in Perl 5 development; however, Jon Orwant suggested (in a particularly vivid and colorful way) that there were deeper problems in the state of Perl and the Perl community that should be fixed by a completely new version of Perl.

The majority consensus was that this was a good idea, and Larry Wall picked up on it. It was presented to the main `perl5-porters` meeting the same afternoon, and various people offered to take roles in the development team. Larry announced the start of Perl 6 development in his keynote “State of the Onion” address the following day.

We then experienced a period of feeling around for the best way to organize the development structure of Perl 6. The single-mailing-list model of Perl 5 was prone to infighting; in addition, the pumpking system was problematic because Perl was beginning to get too big for a single person to maintain, and cases of pumpking burnout were too common.

The consensus was that design should be split between a number of working groups, each of which would have a chair. The first two working groups were `perl6-language` and `perl6-internals`, for language design proposals and implementation design, respectively. The busier working groups spawned subgroups for discussion of more focused topics, and developers were encouraged to express their desires for language change in formal Requests for Changes (RFCs).

The comments stage ended on October 1, 2000, after 361 RFCs were submitted. These went to Larry, who sat down to the grueling task of reading each one to assess its merits. Larry then responded by unfolding the language design in a series of articles called *Apocalypses*. Damian Conway, who through generous sponsorship has been

working full-time for the Perl community, has been assisting Larry, and has also produced explanatory articles called *Exegeses*. This process will continue well into 2002.

On the other side, the Perl 6 internals working group started an almost independent subproject: to write a generic interpreter that could be used for Perl 6, Perl 5, and perhaps other dynamic languages as well. Dan Sugalski volunteered to be the internals designer for this interpreter (codenamed Parrot, after a particularly pervasive April Fool's joke by one of the authors of this book...) and explained his decisions in a series of Parrot Design Documents.

When enough of the design was ready, Simon stepped up to be the release manager in another futile attempt to put off finishing this book. The first public release of Parrot happened on Monday, September 10, 2001.

At the time of this writing, Parrot has support for pluggable data types, both simple and aggregate; it can compile and execute four mini-languages, (mini-Scheme, mini-Perl, and two languages specially written for Parrot: Jako and Cola); it has working and efficient garbage collections; and it has the beginnings of an x86 just-in-time compiler.

You can get the latest release of Parrot from CPAN in Simon's home directory (<http://www.cpan.org/authors/id/S/SI/SIMON/>) or by CVS from the perl.org CVS server (<http://cvs.perl.org/>).

11.4.2 Design and implementation

Dan has been keeping one thing in mind while designing Parrot: speed. The Parrot interpreter will run Perl 6 very fast, and most of the other elements of the design filter down from there. However, we're not forgetting the lessons learned from the Perl 5 internals, and the guts of Parrot are designed to be clearly understandable and easily maintainable.

Parrot deviates from the normal techniques used in building a virtual machine by choosing a register rather than a stack architecture. Although a register-based machine is slightly more difficult to compile for, it has several advantages: first, it lets you use standard compiler optimization techniques tailored for ordinary register-based CPUs; second, it eliminates many of the stack-manipulation operations that take up much of the time of a VM such as Perl 5's; finally, by more closely resembling the underlying hardware, it should be more straightforward to compile down to native code.

Parrot's data abstraction is done via a system of Parrot Magic Cookies (PMCs). These are the equivalent of SVs, but are much more sophisticated. Instead of calling a function on an SV, the PMC carries around with it a vtable (a structure of function pointers) full of the functions it can perform. In a sense, it is an object on which you can call methods. In fact, the PMC abstraction acts as an abstract virtual class, with each language providing vtables that implement the interface; for instance, Perl classes have an addition function that will do the right thing on a Perl value, and Python classes may provide a function that does something different. In this way, the core of Parrot can be language-agnostic, with individual users of Parrot providing data types to fit the needs of their language.

Finally, Parrot has the ability to add in, on a lexically scoped basis, custom ops in addition to its core set. Thus even if a language does certain things wildly differently than Parrot expects, the language will still be able to use the interpreter.

11.4.3 What happens next

Parrot and the design of Perl 6 are developing in parallel; Larry will continue to produce Apocalypses explaining the design, whereas the Parrot hackers are nearing the point where it's worth thinking about compiling real languages onto the VM.

The immediate goals for Parrot at time of writing are to add subroutine and symbol table support, which should be everything needed for a sensible interpreter. By the time the language design firms up, we'll be able to switch emphasis towards writing a compiler from Perl 6 down to Parrot assembler.

11.4.4 The future for Perl 5

If Perl 6 is coming and it's going to be so cool, why have we just written a book about Perl 5? For starters, Perl 6 won't be completed for quite a while—writing a Perl interpreter from scratch is an ambitious exercise! It will also take a long time to become generally accepted.

Perl 5 will continue to be developed up until the release of version 5.8.0, and even then maintenance will continue throughout the lifespan of Perl 6. Perl 5 won't become unsupported.

In short, Perl 5 isn't going away anytime soon. Remember how long it took to get rid of all the Perl 4 interpreters and code? That was when we *wanted* to get rid of it; because Perl 6 is likely to be non-compatible with Perl 5, you can expect uptake to be even slower. There's an awful lot of working Perl 5 code, so people won't want to break it all by upgrading to Perl 6.

11.5 FURTHER READING

More thoughts on patching Perl can be found in the `perl5-porters` FAQ at <http://simon-cozens.org/writings/p5p.faq>, Simon's "So You Want to Be a Perl Porter?" (<http://simon-cozens.org/writings/perlhacktut.html>), and in `pod/perlhack.pod`, `Porting/patching.pod`, and `Porting/pumpking.pod` in the Perl distribution.

11.6 SUMMARY

This chapter looked at how to develop `perl` itself, the development process, and the `perl5-porters` mailing list. In addition to discussing some of the tools available to help you develop, such as `perl`'s debugging mode and the GNU debugger, we also looked at the less technical parts of being a Perl porter—how to approach Perl maintenance, and how to submit patches and get them integrated to the Perl core.

We also discussed Perl 6 and gave you a glimpse of how Perl may look in the future.