NAME

perlembed - how to embed perl in your C program

DESCRIPTION

PREAMBLE

Do you want to:

Use C from Perl?
Read perlxstut, perlxs, h2xs, perlguts, and perlapi.

Use a Unix program from Perl?
Read about back-quotes and about `system` and `exec` in perlfunc.

Use Perl from Perl?
Read about ‘‘do’’ in perlfunc and ‘‘eval’’ in perlfunc and ‘‘require’’ in perlfunc and ‘‘use’’ in perlfunc.

Use C from C?
Rethink your design.

Use Perl from C?
Read on...

ROADMAP

- Compiling your C program
- Adding a Perl interpreter to your C program
- Calling a Perl subroutine from your C program
- Evaluating a Perl statement from your C program
- Performing Perl pattern matches and substitutions from your C program
- Fiddling with the Perl stack from your C program
- Maintaining a persistent interpreter
- Maintaining multiple interpreter instances
- Using Perl modules, which themselves use C libraries, from your C program
- Embedding Perl under Win32

Compiling your C program

If you have trouble compiling the scripts in this documentation, you’re not alone. The cardinal rule: COMPILE THE PROGRAMS IN EXACTLY THE SAME WAY THAT YOUR PERL WAS COMPILED. (Sorry for yelling.)

Also, every C program that uses Perl must link in the *perl library*. What’s that, you ask? Perl is itself written in C; the perl library is the collection of compiled C programs that were used to create your perl executable (*/usr/bin/perl* or equivalent). (Corollary: you can’t use Perl from your C program unless Perl has been compiled on your machine, or installed properly — that’s why you shouldn’t blithely copy Perl executables from machine to machine without also copying the *lib* directory.)

When you use Perl from C, your C program will — usually--allocate, ‘‘run’’, and deallocate a *PerlInterpreter* object, which is defined by the perl library.

If your copy of Perl is recent enough to contain this documentation (version 5.002 or later), then the perl library (and *EXTERN.h* and *perl.h*, which you’ll also need) will reside in a directory that looks like this:

```
/usr/local/lib/perl5/your_architecture_here/CORE
```

or perhaps just

```
/usr/local/lib/perl5/CORE
```

or maybe something like

```
/usr/opt/perl5/CORE
```

Execute this statement for a hint about where to find CORE:

```
perl -MConfig -e 'print $Config{archlib}'
```

Here's how you'd compile the example in the next section, "Adding a Perl interpreter to your C program", on my Linux box:

```
% gcc -O2 -Dbool=char -DHAS_BOOL -I/usr/local/include
-I/usr/local/lib/perl5/i586-linux/5.003/CORE
-L/usr/local/lib/perl5/i586-linux/5.003/CORE
-o interp interp.c -lperl -lm
```

(That's all one line.) On my DEC Alpha running old 5.003_05, the incantation is a bit different:

```
% cc -O2 -Olimit 2900 -DSTANDARD_C -I/usr/local/include
-I/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE
-L/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE -L/usr/local/lib
-D__LANGUAGE_C__ -D_NO_PROTO -o interp interp.c -lperl -lm
```

How can you figure out what to add? Assuming your Perl is post-5.001, execute a `perl -V` command and pay special attention to the "cc" and "ccflags" information.

You'll have to choose the appropriate compiler (*cc*, *gcc*, et al.) for your machine: `perl -MConfig -e 'print $Config{cc}'` will tell you what to use.

You'll also have to choose the appropriate library directory (*/usr/local/lib/...*) for your machine. If your compiler complains that certain functions are undefined, or that it can't locate *-lperl*, then you need to change the path following the `-L`. If it complains that it can' t find *EXTERN.h* and *perl.h*, you need to change the path following the `-I`.

You may have to add extra libraries as well. Which ones? Perhaps those printed by

```
perl -MConfig -e 'print $Config{libs}'
```

Provided your perl binary was properly configured and installed the **ExtUtils::Embed** module will determine all of this information for you:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

If the **ExtUtils::Embed** module isn't part of your Perl distribution, you can retrieve it from http://www.perl.com/perl/CPAN/modules/by-module/ExtUtils/ (If this documentation came from your Perl distribution, then you're running 5.004 or better and you already have it.)

The **ExtUtils::Embed** kit on CPAN also contains all source code for the examples in this document, tests, additional examples and other information you may find useful.

Adding a Perl interpreter to your C program

In a sense, perl (the C program) is a good example of embedding Perl (the language), so I'll demonstrate embedding with *miniperlmain.c*, included in the source distribution. Here's a bastardized, non-portable version of *miniperlmain.c* containing the essentials of embedding:

```
#include <EXTERN.h> /* from the Perl distribution */
#include <perl.h> /* from the Perl distribution */

static PerlInterpreter *my_perl; /*** The Perl interpreter ***/

int main(int argc, char **argv, char **env)
{
PERL_SYS_INIT3(&argc,&argv,&env);
my_perl = perl_alloc();
perl_construct(my_perl);
PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
```

```
perl_run(my_perl);
perl_destruct(my_perl);
perl_free(my_perl);
PERL_SYS_TERM();
}
```

Notice that we don't use the `env` pointer. Normally handed to `perl_parse` as its final argument, `env` here is replaced by `NULL`, which means that the current environment will be used.

The macros *PERL_SYS_INIT3()* and *PERL_SYS_TERM()* provide system-specific tune up of the C runtime environment necessary to run Perl interpreters; they should only be called once regardless of how many interpreters you create or destroy. Call *PERL_SYS_INIT3()* before you create your first interpreter, and *PERL_SYS_TERM()* after you free your last interpreter.

Since *PERL_SYS_INIT3()* may change `env`, it may be more appropriate to provide `env` as an argument to *perl_parse()*.

Also notice that no matter what arguments you pass to *perl_parse()*, *PERL_SYS_INIT3()* must be invoked on the C *main()* argc, argv and env and only once.

Now compile this program (I'll call it *interp.c*) into an executable:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

After a successful compilation, you'll be able to use *interp* just like perl itself:

```
% interp
print "Pretty Good Perl \n";
print "10890 - 9801 is ", 10890 - 9801;
<CTRL-D>
Pretty Good Perl
10890 - 9801 is 1089
```

or

```
% interp -e 'printf("%x", 3735928559)'
deadbeef
```

You can also read and execute Perl statements from a file while in the midst of your C program, by placing the filename in *argv[1]* before calling *perl_run*

Calling a Perl subroutine from your C program

To call individual Perl subroutines, you can use any of the **call_*** functions documented in perlcall. In this example we'll use `call_argv`.

That's shown below, in a program I'll call *showtime.c*.

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
char *args[] = { NULL };
PERL_SYS_INIT3(&argc,&argv,&env);
my_perl = perl_alloc();
perl_construct(my_perl);

perl_parse(my_perl, NULL, argc, argv, NULL);
PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

/*** skipping perl_run() ***/
```

```
call_argv("showtime", G_DISCARD | G_NOARGS, args);

perl_destruct(my_perl);
perl_free(my_perl);
PERL_SYS_TERM();
}
```

where *showtime* is a Perl subroutine that takes no arguments (that's the *G_NOARGS*) and for which I'll ignore the return value (that's the *G_DISCARD*). Those flags, and others, are discussed in perlcall.

I'll define the *showtime* subroutine in a file called *showtime.pl*:

```
print "I shan't be printed.";

sub showtime {
print time;
}
```

Simple enough. Now compile and run:

```
% cc -o showtime showtime.c \
`perl -MExtUtils::Embed -e ccopts -e ldopts`
% showtime showtime.pl
818284590
```

yielding the number of seconds that elapsed between January 1, 1970 (the beginning of the Unix epoch), and the moment I began writing this sentence.

In this particular case we don't have to call *perl_run* as we set the PL_exit_flag PERL_EXIT_DESTRUCT_END which executes END blocks in perl_destruct.

If you want to pass arguments to the Perl subroutine, you can add strings to the `NULL`-terminated `args` list passed to *call_argv*. For other data types, or to examine return values, you'll need to manipulate the Perl stack. That's demonstrated in ''Fiddling with the Perl stack from your C program''.

Evaluating a Perl statement from your C program

Perl provides two API functions to evaluate pieces of Perl code. These are ''eval_sv'' in perlapi(1) and ''eval_pv'' in perlapi.

Arguably, these are the only routines you'll ever need to execute snippets of Perl code from within your C program. Your code can be as long as you wish; it can contain multiple statements; it can employ ''use'' in perlfunc(1), ''require'' in perlfunc(1), and ''do'' in perlfunc(1) to include external Perl files.

eval_pv lets us evaluate individual Perl strings, and then extract variables for coercion into C types. The following program, *string.c*, executes three Perl strings, extracting an `int` from the first, a `float` from the second, and a `char *` from the third.

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

main (int argc, char **argv, char **env)
{
char *embedding[] = { "", "-e", "0" };

PERL_SYS_INIT3(&argc,&argv,&env);
my_perl = perl_alloc();
perl_construct( my_perl );

perl_parse(my_perl, NULL, 3, embedding, NULL);
```

```
PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
perl_run(my_perl);

/** Treat $a as an integer **/
eval_pv("$a = 3; $a **= 2", TRUE);
printf("a = %d\n", SvIV(get_sv("a", 0)));

/** Treat $a as a float **/
eval_pv("$a = 3.14; $a **= 2", TRUE);
printf("a = %f\n", SvNV(get_sv("a", 0)));

/** Treat $a as a string **/
eval_pv(
"$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($a);", TRUE);
printf("a = %s\n", SvPV_nolen(get_sv("a", 0)));

perl_destruct(my_perl);
perl_free(my_perl);
PERL_SYS_TERM();
}
```

All of those strange functions with *sv* in their names help convert Perl scalars to C types. They're described in perlguts(1) and perlapi.

If you compile and run *string.c*, you'll see the results of using *SvIV()* to create an `int`, *SvNV()* to create a `float`, and *SvPV()* to create a string:

```
a = 9
a = 9.859600
a = Just Another Perl Hacker
```

In the example above, we've created a global variable to temporarily store the computed value of our eval'ed expression. It is also possible and in most cases a better strategy to fetch the return value from *eval_pv()* instead. Example:

```
...
SV *val = eval_pv("reverse 'rekcaH lreP rehtonA tsuJ'", TRUE);
printf("%s\n", SvPV_nolen(val));
...
```

This way, we avoid namespace pollution by not creating global variables and we've simplified our code as well.

Performing Perl pattern matches and substitutions from your C program

The *eval_sv()* function lets us evaluate strings of Perl code, so we can define some functions that use it to "specialize" in matches and substitutions: *match()*, *substitute()*, and *matches()*.

```
I32 match(SV *string, char *pattern);
```

Given a string and a pattern (e.g., `m/clasp/` or `/\b\w*\b/`, which in your C program might appear as "/\\b\\w*\\b/"), *match()* returns 1 if the string matches the pattern and 0 otherwise.

```
int substitute(SV **string, char *pattern);
```

Given a pointer to an SV and an `=~` operation (e.g., `s/bob/robert/g` or `tr[A-Z][a-z]`), *substitute()* modifies the string within the SV as according to the operation, returning the number of substitutions made.

```
SSize_t matches(SV *string, char *pattern, AV **matches);
```

Given an SV, a pattern, and a pointer to an empty AV, *matches()* evaluates `$string =~ $pattern` in a list context, and fills in *matches* with the array elements, returning the number of matches found.

Here's a sample program, *match.c*, that uses all three (long lines have been wrapped here):

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

/** my_eval_sv(code, error_check)
** kinda like eval_sv(),
** but we pop the return value off the stack
**/
SV* my_eval_sv(SV *sv, I32 croak_on_error)
{
dSP;
SV* retval;


PUSHMARK(SP);
eval_sv(sv, G_SCALAR);

SPAGAIN;
retval = POPs;
PUTBACK;

if (croak_on_error && SvTRUE(ERRSV))
croak(SvPVx_nolen(ERRSV));

return retval;
}

/** match(string, pattern)
**
** Used for matches in a scalar context.
**
** Returns 1 if the match was successful; 0 otherwise.
**/

I32 match(SV *string, char *pattern)
{
SV *command = newSV(0), *retval;

sv_setpvf(command, "my $string = '%s'; $string =~ %s",
SvPV_nolen(string), pattern);

retval = my_eval_sv(command, TRUE);
SvREFCNT_dec(command);

return SvIV(retval);
}

/** substitute(string, pattern)
**
** Used for =~ operations that
** modify their left-hand side (s/// and tr///)
**
** Returns the number of successful matches, and
```

```
** modifies the input string if there were any.
**/

I32 substitute(SV **string, char *pattern)
{
SV *command = newSV(0), *retval;

sv_setpvf(command, "$string = '%s'; ($string =~ %s)",
SvPV_nolen(*string), pattern);

retval = my_eval_sv(command, TRUE);
SvREFCNT_dec(command);

*string = get_sv("string", 0);
return SvIV(retval);
}

/** matches(string, pattern, matches)
**
** Used for matches in a list context.
**
** Returns the number of matches,
** and fills in **matches with the matching substrings
**/

SSize_t matches(SV *string, char *pattern, AV **match_list)
{
SV *command = newSV(0)
SSize_t num_matches;

sv_setpvf(command, "my $string = '%s'; @array = ($string =~ %s)",
SvPV_nolen(string), pattern);

my_eval_sv(command, TRUE);
SvREFCNT_dec(command);

*match_list = get_av("array", 0);
num_matches = av_top_index(*match_list) + 1;

return num_matches;
}

main (int argc, char **argv, char **env)
{
char *embedding[] = { "", "-e", "0" };
AV *match_list;
I32 num_matches, i;
SV *text;

PERL_SYS_INIT3(&argc,&argv,&env);
my_perl = perl_alloc();
perl_construct(my_perl);
perl_parse(my_perl, NULL, 3, embedding, NULL);
PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
```

```
text = newSV(0)
sv_setpv(text, "When he is at a convenience store and the "
"bill comes to some amount like 76 cents, Maynard is "
"aware that there is something he *should* do, something "
"that will enable him to get back a quarter, but he has "
"no idea *what*. He fumbles through his red squeezey "
"changepurse and gives the boy three extra pennies with "
"his dollar, hoping that he might luck into the correct "
"amount. The boy gives him back two of his own pennies "
"and then the big shiny quarter that is his prize. "
"-RICHH");

if (match(text, "m/quarter/")) /** Does text contain 'quarter'? **/
printf("match: Text contains the word 'quarter'.\n\n");
else
printf("match: Text doesn't contain the word 'quarter'.\n\n");

if (match(text, "m/eighth/")) /** Does text contain 'eighth'? **/
printf("match: Text contains the word 'eighth'.\n\n");
else
printf("match: Text doesn't contain the word 'eighth'.\n\n");

/** Match all occurrences of /wi../ **/
num_matches = matches(text, "m/(wi..)/g", &match_list);
printf("matches: m/(wi..)/g found %d matches...\n", num_matches);

for (i = 0; i < num_matches; i++)
printf("match: %s\n",
SvPV_nolen(*av_fetch(match_list, i, FALSE)));
printf("\n");

/** Remove all vowels from text **/
num_matches = substitute(&text, "s/[aeiou]//gi");
if (num_matches) {
printf("substitute: s/[aeiou]//gi...%lu substitutions made.\n",
(unsigned long)num_matches);
printf("Now text is: %s\n\n", SvPV_nolen(text));
}

/** Attempt a substitution **/
if (!substitute(&text, "s/Perl/C/")) {
printf("substitute: s/Perl/C...No substitution made.\n\n");
}

SvREFCNT_dec(text);
PL_perl_destruct_level = 1;
perl_destruct(my_perl);
perl_free(my_perl);
PERL_SYS_TERM();
}
```

which produces the output (again, long lines have been wrapped here)

```
match: Text contains the word 'quarter'.
```

```
match: Text doesn't contain the word 'eighth'.

matches: m/(wi..)/g found 2 matches...
match: will
match: with

substitute: s/[aeiou]//gi...139 substitutions made.
Now text is: Whn h s t cnvnnc str nd th bll cms t sm mnt lk 76 cnts,
Mynrd s wr tht thr s smthng h *shld* d, smthng tht wll nbl hm t gt
bck qrtr, bt h hs n d *wht*. H fmbls thrgh hs rd sqzy chngprs nd
gvs th by thr xtr pnns wth hs dllr, hpng tht h mght lck nt th crrct
mnt. Th by gvs hm bck tw f hs wn pnns nd thn th bg shny qrtr tht s
hs prz. -RCHH

substitute: s/Perl/C...No substitution made.
```

Fiddling with the Perl stack from your C program

When trying to explain stacks, most computer science textbooks mumble something about spring-loaded columns of cafeteria plates: the last thing you pushed on the stack is the first thing you pop off. That'll do for our purposes: your C program will push some arguments onto ''the Perl stack'', shut its eyes while some magic happens, and then pop the results — the return value of your Perl subroutine — off the stack.

First you'll need to know how to convert between C types and Perl types, with *newSViv()* and *sv_setnv()* and *newAV()* and all their friends. They're described in perlguts(1) and perlapi.

Then you'll need to know how to manipulate the Perl stack. That's described in perlcall.

Once you've understood those, embedding Perl in C is easy.

Because C has no builtin function for integer exponentiation, let's make Perl's ** operator available to it (this is less useful than it sounds, because Perl implements ** with C's *pow()* function). First I'll create a stub exponentiation function in *power.pl*:

```
sub expo {
my ($a, $b) = @_;
return $a ** $b;
}
```

Now I'll create a C program, *power.c*, with a function *PerlPower()* that contains all the perlguts(1) necessary to push the two arguments into *expo()* and to pop the return value out. Take a deep breath...

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

static void
PerlPower(int a, int b)
{
dSP; /* initialize stack pointer */
ENTER; /* everything created after here */
SAVETMPS; /* ...is a temporary variable. */
PUSHMARK(SP); /* remember the stack pointer */
XPUSHs(sv_2mortal(newSViv(a))); /* push the base onto the stack */
XPUSHs(sv_2mortal(newSViv(b))); /* push the exponent onto stack */
PUTBACK; /* make local stack pointer global */
call_pv("expo", G_SCALAR); /* call the function */
SPAGAIN; /* refresh stack pointer */
/* pop the return value from stack */
```

```
printf ("%d to the %dth power is %d.\n", a, b, POPi);
PUTBACK;
FREETMPS; /* free that return value */
LEAVE; /* ...and the XPUSHed "mortal" args.*/
}

int main (int argc, char **argv, char **env)
{
char *my_argv[] = { "", "power.pl" };

PERL_SYS_INIT3(&argc,&argv,&env);
my_perl = perl_alloc();
perl_construct( my_perl );

perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
perl_run(my_perl);

PerlPower(3, 4); /*** Compute 3 ** 4 ***/

perl_destruct(my_perl);
perl_free(my_perl);
PERL_SYS_TERM();
}
```

Compile and run:

```
% cc -o power power.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% power
3 to the 4th power is 81.
```

Maintaining a persistent interpreter

When developing interactive and/or potentially long-running applications, it's a good idea to maintain a persistent interpreter rather than allocating and constructing a new interpreter multiple times. The major reason is speed: since Perl will only be loaded into memory once.

However, you have to be more cautious with namespace and variable scoping when using a persistent interpreter. In previous examples we've been using global variables in the default package `main`. We knew exactly what code would be run, and assumed we could avoid variable collisions and outrageous symbol table growth.

Let's say your application is a server that will occasionally run Perl code from some arbitrary file. Your server has no way of knowing what code it's going to run. Very dangerous.

If the file is pulled in by `perl_parse()`, compiled into a newly constructed interpreter, and subsequently cleaned out with `perl_destruct()` afterwards, you're shielded from most namespace troubles.

One way to avoid namespace collisions in this scenario is to translate the filename into a guaranteed-unique package name, and then compile the code into that package using ''eval'' in perlfunc. In the example below, each file will only be compiled once. Or, the application might choose to clean out the symbol table associated with the file after it's no longer needed. Using ''call_argv'' in perlapi(1), We'll call the subroutine `Embed::Persistent::eval_file` which lives in the file `persistent.pl` and pass the filename and boolean cleanup/cache flag as arguments.

Note that the process will continue to grow for each file that it uses. In addition, there might be `AUTOLOAD`ed subroutines and other conditions that cause Perl's symbol table to grow. You might want to add some logic that keeps track of the process size, or restarts itself after a certain number of requests, to ensure that memory consumption is minimized. You'll also want to scope your variables with ''my'' in

perlfunc(1) whenever possible.

```
package Embed::Persistent;
#persistent.pl

use strict;
our %Cache;
use Symbol qw(delete_package);

sub valid_package_name {
my($string) = @_;
$string =~ s/([^A-Za-z0-9\/])/sprintf("_%2x",unpack("C",$1))/eg;
# second pass only for words starting with a digit
$string =~ s|/(\d)|sprintf("/_%2x",unpack("C",$1))|eg;

# Dress it up as a real package name
$string =~ s|/|::|g;
return "Embed" . $string;
}

sub eval_file {
my($filename, $delete) = @_;
my $package = valid_package_name($filename);
my $mtime = -M $filename;
if(defined $Cache{$package}{mtime}
&&
$Cache{$package}{mtime} <= $mtime)
{
# we have compiled this subroutine already,
# it has not been updated on disk, nothing left to do
print STDERR "already compiled $package->handler\n";
}
else {
local *FH;
open FH, $filename or die "open '$filename' $!";
local($/) = undef;
my $sub = <FH>;
close FH;

#wrap the code into a subroutine inside our unique package
my $eval = qq{package $package; sub handler { $sub; }};
{
# hide our variables within this block
my($filename,$mtime,$package,$sub);
eval $eval;
}
die $@ if $@;

#cache it unless we're cleaning out each time
$Cache{$package}{mtime} = $mtime unless $delete;
}

eval {$package->handler;};
die $@ if $@;
```

```
delete_package($package) if $delete;

#take a look if you want
#print Devel::Symdump->rnew($package)->as_string, $/;
}

1;

__END__

/* persistent.c */
#include <EXTERN.h>
#include <perl.h>

/* 1 = clean out filename's symbol table after each request,
0 = don't
*/
#ifndef DO_CLEAN
#define DO_CLEAN 0
#endif

#define BUFFER_SIZE 1024

static PerlInterpreter *my_perl = NULL;

int
main(int argc, char **argv, char **env)
{
char *embedding[] = { "", "persistent.pl" };
char *args[] = { "", DO_CLEAN, NULL };
char filename[BUFFER_SIZE];
int exitstatus = 0;

PERL_SYS_INIT3(&argc,&argv,&env);
if((my_perl = perl_alloc()) == NULL) {
fprintf(stderr, "no memory!");
exit(1)
}
perl_construct(my_perl);

PL_origalen = 1; /* don't let $0 assignment update the
proctitle or embedding[0] */
exitstatus = perl_parse(my_perl, NULL, 2, embedding, NULL);
PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
if(!exitstatus) {
exitstatus = perl_run(my_perl);

while(printf("Enter file name: ") &&
fgets(filename, BUFFER_SIZE, stdin)) {

filename[strlen(filename)-1] = '\0'; /* strip \n */
/* call the subroutine,
passing it the filename as an argument */
args[0] = filename;
```

```
call_argv("Embed::Persistent::eval_file",
G_DISCARD | G_EVAL, args);

/* check $@ */
if(SvTRUE(ERRSV))
fprintf(stderr, "eval error: %s\n", SvPV_nolen(ERRSV));
}
}

PL_perl_destruct_level = 0;
perl_destruct(my_perl);
perl_free(my_perl);
PERL_SYS_TERM();
exit(exitstatus);
}
```

Now compile:

```
% cc -o persistent persistent.c \
`perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Here's an example script file:

```
#test.pl
my $string = "hello";
foo($string);

sub foo {
print "foo says: @_\n";
}
```

Now run:

```
% persistent
Enter file name: test.pl
foo says: hello
Enter file name: test.pl
already compiled Embed::test_2epl->handler
foo says: hello
Enter file name: ^C
```

Execution of END blocks

Traditionally END blocks have been executed at the end of the perl_run. This causes problems for applications that never call perl_run. Since perl 5.7.2 you can specify `PL_exit_flags |= PERL_EXIT_DESTRUCT_END` to get the new behaviour. This also enables the running of END blocks if the perl_parse fails and `perl_destruct` will return the exit value.

`$0` assignments

When a perl script assigns a value to `$0` then the perl runtime will try to make this value show up as the program name reported by ''ps'' by updating the memory pointed to by the argv passed to *perl_parse()* and also calling API functions like *setproctitle()* where available. This behaviour might not be appropriate when embedding perl and can be disabled by assigning the value `1` to the variable `PL_origalen` before *perl_parse()* is called.

The *persistent.c* example above is for instance likely to segfault when `$0` is assigned to if the `PL_origalen = 1;` assignment is remo ved. This because perl will try to write to the read only memory of the `embedding[]` strings.

Maintaining multiple interpreter instances

Some rare applications will need to create more than one interpreter during a session. Such an application might sporadically decide to release any resources associated with the interpreter.

The program must take care to ensure that this takes place *before* the next interpreter is constructed. By default, when perl is not built with any special options, the global variable `PL_perl_destruct_level` is set to `0`, since extra cleaning isn't usually needed when a program only ever creates a single interpreter in its entire lifetime.

Setting `PL_perl_destruct_level` to `1` makes everything squeaky clean:

```
while(1) {
...
/* reset global variables here with PL_perl_destruct_level = 1 */
PL_perl_destruct_level = 1;
perl_construct(my_perl);
...
/* clean and reset _everything_ during perl_destruct */
PL_perl_destruct_level = 1;
perl_destruct(my_perl);
perl_free(my_perl);
...
/* let's go do it again! */
}
```

When *perl_destruct()* is called, the interpreter's syntax parse tree and symbol tables are cleaned up, and global variables are reset. The second assignment to `PL_perl_destruct_level` is needed because perl_construct resets it to `0`.

Now suppose we have more than one interpreter instance running at the same time. This is feasible, but only if you used the Configure option `-Dusemultiplicity` or the options `-Dusethreads -Duseithreads` when building perl. By default, enabling one of these Configure options sets the per-interpreter global variable `PL_perl_destruct_level` to `1`, so that thorough cleaning is automatic and interpreter variables are initialized correctly. Even if you don't intend to run two or more interpreters at the same time, but to run them sequentially, like in the above example, it is recommended to build perl with the `-Dusemultiplicity` option otherwise some interpreter variables may not be initialized correctly between consecutive runs and your application may crash.

See also ‘‘Thread-aware system interfaces’’ in perlxs.

Using `-Dusethreads -Duseithreads` rather than `-Dusemultiplicity` is more appropriate if you intend to run multiple interpreters concurrently in different threads, because it enables support for linking in the thread libraries of your system with the interpreter.

Let's give it a try:

```
#include <EXTERN.h>
#include <perl.h>

/* we're going to embed two interpreters */

#define SAY_HELLO "-e", "print qq(Hi, I'm $^X\n)"

int main(int argc, char **argv, char **env)
{
PerlInterpreter *one_perl, *two_perl;
char *one_args[] = { "one_perl", SAY_HELLO };
char *two_args[] = { "two_perl", SAY_HELLO };

PERL_SYS_INIT3(&argc,&argv,&env);
```

```
one_perl = perl_alloc();
two_perl = perl_alloc();

PERL_SET_CONTEXT(one_perl);
perl_construct(one_perl);
PERL_SET_CONTEXT(two_perl);
perl_construct(two_perl);

PERL_SET_CONTEXT(one_perl);
perl_parse(one_perl, NULL, 3, one_args, (char **)NULL);
PERL_SET_CONTEXT(two_perl);
perl_parse(two_perl, NULL, 3, two_args, (char **)NULL);

PERL_SET_CONTEXT(one_perl);
perl_run(one_perl);
PERL_SET_CONTEXT(two_perl);
perl_run(two_perl);

PERL_SET_CONTEXT(one_perl);
perl_destruct(one_perl);
PERL_SET_CONTEXT(two_perl);
perl_destruct(two_perl);

PERL_SET_CONTEXT(one_perl);
perl_free(one_perl);
PERL_SET_CONTEXT(two_perl);
perl_free(two_perl);
PERL_SYS_TERM();
}
```

Note the calls to *PERL_SET_CONTEXT()*. These are necessary to initialize the global state that tracks which interpreter is the ‘‘current’’ one on the particular process or thread that may be running it. It should always be used if you have more than one interpreter and are making perl API calls on both interpreters in an interleaved fashion.

PERL_SET_CONTEXT(interp) should also be called whenever `interp` is used by a thread that did not create it (using either *perl_alloc()*, or the more esoteric *perl_clone()*).

Compile as usual:

```
% cc -o multiplicity multiplicity.c \
`perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Run it, Run it:

```
% multiplicity
Hi, I'm one_perl
Hi, I'm two_perl
```

Using Perl modules, which themselves use C libraries, from your C program

If you’ve played with the examples above and tried to embed a script that *use()*s a Perl module (such as *Socket*) which itself uses a C or C++ library, this probably happened:

```
Can't load module Socket, dynamic loading not available in this perl.
(You may need to build a new perl executable which either supports
dynamic loading or has the Socket module statically linked into it.)
```

What’s wrong?

Your interpreter doesn’t know how to communicate with these extensions on its own. A little glue will help.

Up until now you've been calling *perl_parse()*, handing it NULL for the second argument:

```
perl_parse(my_perl, NULL, argc, my_argv, NULL);
```

That's where the glue code can be inserted to create the initial contact between Perl and linked C/C++ routines. Let's take a look some pieces of *perlmain.c* to see how Perl does this:

```
static void xs_init (pTHX);

EXTERN_C void boot_DynaLoader (pTHX_ CV* cv);
EXTERN_C void boot_Socket (pTHX_ CV* cv);


EXTERN_C void
xs_init(pTHX)
{
char *file = __FILE__;
/* DynaLoader is a special case */
newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
newXS("Socket::bootstrap", boot_Socket, file);
}
```

Simply put: for each extension linked with your Perl executable (determined during its initial configuration on your computer or when adding a new extension), a Perl subroutine is created to incorporate the extension's routines. Normally, that subroutine is named *Module::bootstrap()* and is invoked when you say *use Module*. In turn, this hooks into an XSUB, *boot_Module*, which creates a Perl counterpart for each of the extension's XSUBs. Don't worry about this part; leave that to the *xsubpp* and extension authors. If your extension is dynamically loaded, DynaLoader creates *Module::bootstrap()* for you on the fly. In fact, if you have a working DynaLoader then there is rarely any need to link in any other extensions statically.

Once you have this code, slap it into the second argument of *perl_parse()*:

```
perl_parse(my_perl, xs_init, argc, my_argv, NULL);
```

Then compile:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% interp
use Socket;
use SomeDynamicallyLoadedModule;

print "Now I can use extensions!\n"'
```

ExtUtils::Embed can also automate writing the *xs_init* glue code.

```
% perl -MExtUtils::Embed -e xsinit -- -o perlxsi.c
% cc -c perlxsi.c `perl -MExtUtils::Embed -e ccopts`
% cc -c interp.c `perl -MExtUtils::Embed -e ccopts`
% cc -o interp perlxsi.o interp.o `perl -MExtUtils::Embed -e ldopts`
```

Consult perlxs(1), perlguts(1), and perlapi(1) for more details.

Using embedded Perl with POSIX locales

(See perllocale(1) for information about these.) When a Perl interpreter normally starts up, it tells the system it wants to use the system's default locale. This is often, but not necessarily, the ''C'' or ''POSIX'' locale. Absent a `"uselocale"` within the perl code, this mostly has no effect (but see ''Not within the scope of ''use locale"" in perllocale). Also, there is not a problem if the locale you want to use in your embedded Perl is the same as the system default. However, this doesn't work if you have set up and want to use a locale that isn't the system default one. Starting in Perl v5.20, you can tell the embedded Perl interpreter that the locale is already properly set up, and to skip doing its own normal initialization. It skips if the environment variable `PERL_SKIP_LOCALE_INIT` is set (even if set to 0 or `""`). A Perl that has

this capability will define the C pre-processor symbol `HAS_SKIP_LOCALE_INIT`. This allows code that has to work with multiple Perl versions to do some sort of work-around when confronted with an earlier Perl.

Hiding Perl_

If you completely hide the short forms of the Perl public API, add -DPERL_NO_SHORT_NAMES to the compilation flags. This means that for example instead of writing

```
warn("%d bottles of beer on the wall", bottlecount);
```

you will have to write the explicit full form

```
Perl_warn(aTHX_ "%d bottles of beer on the wall", bottlecount);
```

(See "Background and PERL_IMPLICIT_CONTEXT" in perlguts(1) for the explanation of the `aTHX_`.) Hiding the short forms is very useful for avoiding all sorts of nasty (C preprocessor or otherwise) conflicts with other software packages (Perl defines about 2400 APIs with these short names, take or leave few hundred, so there certainly is room for conflict.)

MORAL

You can sometimes *write faster code* in C, but you can always *write code faster* in Perl. Because you can use each from the other, combine them as you wish.

AUTHOR

Jon Orwant <*orwant@media.mit.edu*> and Doug MacEachern <*dougm@covalent.net*>, with small contributions from Tim Bunce, Tom Christiansen, Guy Decoux, Hallvard Furuseth, Dov Grobgeld, and Ilya Zakharevich.

Doug MacEachern has an article on embedding in Volume 1, Issue 4 of The Perl Journal (http://www.tpj.com/). Doug is also the developer of the most widely-used Perl embedding: the mod_perl system (perl.apache.org), which embeds Perl in the Apache web server. Oracle, Binary Evolution, ActiveState, and Ben Sugars's nsapi_perl have used this model for Oracle, Netscape and Internet Information Server Perl plugins.

COPYRIGHT