Embedding Perl in C

8.1 When to embed 254
8.2 When not to embed 255
8.3 Things to think about 255
8.4 “Hello C” from Perl 255
8.5 Passing data 257
8.6 Calling Perl routines 259
8.7 Using C in Perl in C 265
8.8 Embedding wisdom 266
8.9 Summary 267

In the first half of this book, we’ve looked at what it means to extend Perl with additional routines from C. Sometimes, however, you’ll want to call a piece of Perl from inside a C program—this is called *embedding* Perl in C, because you link an entire Perl interpreter inside another C program.

### 8.1 When to Embed

The best and most well-known example of embedding Perl in C is Apache’s `mod_perl` module. It allows the user to interact with Perl at every level of the Apache web server—you can write configuration files in Perl, write Perl programs to handle HTTP requests with Perl objects, and so on. In short, it lets you use Perl to script and control the rest of the program.

More specific examples include the embedding of Perl into the Xchat IRC client to enable the user to script complex actions in Perl; the GIMP graphics editor, which allows graphical manipulations to be encoded in Perl; the `vim` text editor, which can be both configured and manipulated using Perl; and `gnumeric`, the GNOME spreadsheet, which exposes the data in the spreadsheet cells to Perl for additional manipulation.
All of these examples have common objectives:

- To make the application extensible through user-provided plug-in scripts
- To make configuration more flexible by involving a high-level language and the control structures it provides
- To help the user script common or complex sequences of actions

If an application you are working with could benefit from these features, you should contemplate embedding a Perl interpreter.

8.2 **WHEN NOT TO EMBED**

Embedding Perl into a program is not a panacea, and embedding Perl into an existing program is not a step to be taken lightly. We don't recommend embedded Perl as a cheap way of avoiding writing a configuration parser or extensible scripting system.

You should also be conscious that embedding Perl into an application will increase its size and memory usage, possibly introduce memory leaks or instabilities if you aren't careful, and occasionally slow the application. Nevertheless, the examples we have given show that, in a lot of cases, you can make a big gain by including Perl in your program.

8.3 **THINGS TO THINK ABOUT**

In chapter 9, we'll look in more detail at the decisions you need to make when embedding Perl into an application. Fundamentally, however, you must consider the degree to which Perl should have access to the guts of your program.

This consideration in turn influences details such as which data structures you will expose to Perl and how they will appear to the Perl programmer; which C functions in your API will be available, and, again, how they would be used from Perl; where your Perl programs will come from and at what point in the program they will be used; and so on.

Again, you'll see practical answers to these questions in chapter 9. Let's now look at an example of calling a Perl program from inside a C program.

8.4 **“HELLO C” FROM PERL**

The fundamentals of embedding are simple: you perform almost exactly the same function as the main body of the `perl` binary. That is, you construct and initialize an interpreter, use it to parse a string of code, and then execute that code. Here's a simple program that does so:

```c
#include <EXTERN.h>  
#include <perl.h>  
static PerlInterpreter *my_perl;  
int main(int argc, char **argv)  
```

This first header file sets up some macros to tell the main Perl header file that we're not really Perl, but an external application using the Perl API.

We load the main Perl header, which provides the macros and prototypes for all the functions in the Perl API.

It is possible to have multiple Perl interpreters in a program, because all the interpreter-global data is stored in a structure called, naturally, a PerlInterpreter. In this program we have one interpreter, which is stored in the pointer my_perl.

Even when we're embedding Perl, we're dealing with the honest-to-goodness Perl interpreter, which expects to get arguments from the command line. Hence, we must provide a set of command-line arguments, just as we'd expect to find in argv. (And just like in argv, the first element of the array is the name of the command, rather than the first command-line argument; we're not bothered about the name of the command, so we leave it blank.)

We need to allocate memory, just as we would for any other structure pointer; perl_alloc returns some memory for a Perl interpreter.

Next, we set up the interpreter and all its associated data structures with perl_construct.

Now we're in a position where we can parse the incoming Perl “script,” which is specified in the -e argument to the faked command line.

Perl is bytecode-compiled—first the code is parsed into an internal representation, and then it's run. perl_run starts the main loop running the code.

We cleanly shut down the interpreter and release the memory that had been allocated for it.

It's interesting to compare this code with the source of the Perl interpreter; if you don't believe we're performing the same functions as the Perl interpreter, take a look at the guts of miniperlmain.c from Perl 5.6.1:
if (!PL_do_undump) {
    my_perl = perl_alloc();
    if (!my_perl)
        exit(1);
    perl_construct(my_perl);
    PL_perl_destruct_level = 0;
}

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

perl_destruct(my_perl);
perl_free(my_perl);
PERL_SYS_TERM();
exit(exitstatus);
return exitstatus;

As you can see, this is much the same as our example, with a little more error checking.

8.4.1 Compiling embedded programs

Compiling programs that embed Perl is a little special; you must ensure that you’re compiling the code with exactly the same options that were used to compile the Perl interpreter. Although you could get them from Config.pm, a module makes it easy for you: ExtUtils::Embed.

As you know from chapter 1, a program’s compilation takes place in two stages: compilation proper and linking. ExtUtils::Embed provides two functions, ccopts and ldopts, to tell you the options for each stage. If you run these functions from the perl command line, they’ll handily spit out the options to standard output, making the module ideal to use as part of your build process.

Let’s compile and then link the previous example code, simple.c:

```bash
% cc -o simple.o -c simple.c `perl -MExtUtils::Embed -e ccopts`
% cc -o simple simple.o `perl -MExtUtils::Embed -e ldopts`
% ./simple
Hello from C!
```

Now you have a way to execute simple Perl programs from C, albeit if you specify them on the Perl command line. Let’s make this approach a bit more powerful.

8.5 PASSING DATA

In a real embedded application, you need a way to share data between Perl and C. For instance, the Apache embedded Perl module, mod_perl, allows you to store configuration data in Perl variables.
As you saw on "get_sv," page 103, Perl provides a function called get_sv that lets you grab an SV from the Perl symbol table. Suppose we're writing a mail client (we'll call it Hermes) and we want our users to be able to set some of the configuration in Perl. First, we'll look at general settings that apply to the whole application; in the next section, we'll write some logic for settings that apply on a per-message basis.

Our sample configuration file looks like this:

```
package Hermes;
$save_outgoing = 1;
# Prefer vim, but use emacs if vim not available.
$editor = `which vim` || `which emacs`;
$quote_char = " > ";
$check_every = 10; # seconds

Inside our mail client, we'll have Perl parse and run this configuration; we also want to get at the results. We know how to do the first part—we allocate, make, and instantiate a Perl interpreter:

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

static PerlInterpreter *my_perl;

int parse_config(char * config_file)
{
    char* command_line[2] = {"", NULL};
    command_line[1] = config_file;
    my_perl = perl_alloc();
    perl_construct(my_perl);
    if (perl_parse(my_perl, NULL, 2, command_line, (char **)NULL)) {
        return 0; /* Failed to parse */
    }
    perl_run(my_perl);
    if (SvTRUE(ERRSV)) {
        return 0; /* Failed to execute */
    }
    return 1;
}
```

This code is substantially the same as the previous code, except that the "command line" passed to Perl is determined at runtime when the name of the configuration file is passed in to parse_config. The other difference is that once we have run the Perl program, we check whether ERRSV is true; this is the C-side equivalent of checking $@. We also don't destroy the interpreter, because we'll be using it to get at Perl values and execute Perl code in the rest of our mailer.
Now we have executed the Perl program, and we should be able to determine the values of the configuration variables using the get_sv function. For instance, suppose it is time to edit an email message before sending it; we look up the location of the editor we're going to use:

```c
int edit_message(char* filename) {
    char *editor;
    editor = SvPV_nolen(get_sv("Hermes::editor"));
    /* And now we execute the editor */
    ...
}
```

Similarly, we can set these SVs to values from our C code using sv_setpv or sv_setiv if we want to communicate information back to Perl. However, you'll usually pass values to Perl subroutines; let's see how to do this.

### 8.6 Calling Perl Routines

The techniques you saw in section 6.7.1 for calling Perl subroutines from XS are equally applicable to calling Perl subroutines from an embedded program. You still need to put parameters onto the stack, make a call out to Perl, and collect return values. Let's take a closer look at those techniques.

Perl has a number of different functions for calling routines, but the two we'll concentrate on are call_sv and call_pv. It's easy to decide which one to call: if you have an SV that refers to a subroutine—one that contains either a code reference or a subroutine's name (a symbolic reference)—then you use call_sv. Otherwise, if you only have the name as a string, you use call_pv. The typical embedded program will generally call subroutines by name using call_pv, although there are instances where call_sv is correct.

For example, suppose we want to allow users of our mailer to define a subroutine that is passed a mail message for preprocessing as the user replies to it. We could state in our program's embedding API that the configuration file must define a subroutine called Hermes::reply_to_hook, and we could use call_pv with the string "Hermes::reply_to_hook"; or, we could allow something like this:

```perl
package Hermes;
$reply_to_hook = sub {
    my $mail = @_; # your code here
    ...
};
```

In this case, we'd use get_sv("Hermes::reply_to_hook", TRUE) to return a code reference, which we'd then call with call_sv.

You'll see another example where call_sv is necessary when we look at callbacks later in this section. For now, we'll concentrate on call_pv.
Here’s the simplest possible instance of calling a Perl routine from C; it has no parameters and no return value. One good example of this type of routine would be an exit handler—something that our mail program calls when it’s about to quit, so that any state established by the Perl program can be cleared up.¹ This would be the Perl side of it:

```perl
package Hermes;
sub atexit {
    # Close any open file handles
    ...
    #
    print "Thank you for using Hermes/Perl. Going away now.\n";
}
```

And here is the C side:

```c
/* Clean up Perl embedding */
void perl_stop(void) {
    dSP;  // Provides local access to Perl's argument stack
    PUSHMARK(SP);  // Pushes a bookmark onto the argument stack. In our example in chapter 6, we used a complicated prologue to call the callback, using ENTER/SAVETMPS/PUSHMARK and PUTBACK/FREETMPS/LEAVE; but we are not concerned about parameters and return values here, so we can do without most of these macros. We must still push a bookmark, because the argument stack is not always empty; if we’re already inside a Perl expression, the stack will not be empty. However, when a callback receives parameters or returns values, it needs to know how many items from the top of the stack belong to it. For instance, there may already be four items on the stack before we call Hermes::atexit. Suppose we want to push another two items on as parameters. Perl needs some way to know that the four items previously on the

¹ Of course, this is what END blocks are for. If we used an END block in our Perl configuration file, then all these steps would be done automatically—but that would not make a good example.
Now we call the routine. \texttt{call_pv} and \texttt{call_sv} take two parameters: the name or \texttt{SV} for the subroutine to be called, and a set of flags. In this instance, our flags are \texttt{G_DISCARD}, signifying that we're going to discard the return value of the call (and therefore that \texttt{Hermes::atexit} should be called in void context); and \texttt{G_NOARGS}, to state that we're not pushing in any arguments, so the subroutine doesn't need to look at the stack.

We've called the routine, so we can shut down the interpreter and free the memory it used, in preparation for exiting the program.

8.6.1 Stack manipulation

So much for the easy case. Unfortunately, in most other examples, you must deal with both parameters and return values.

\textbf{Parameters}

You already know the basics of handling parameters from “Returning a list,” page 181; you use \texttt{PUSHs} to put things on the stack. To be sure the values you put onto the stack are properly garbage collected, you need to make them temporary; doing so tells Perl that they should go away at the end of the scope. Thus you must declare a scope—and this is where the \texttt{ENTER/SAVETMPS/PUSHMARK} business comes in.

Let's take another example from our mythical mailer and pass a mail message to a Perl function for preprocessing. The message will come in as an array of strings, so we need to make them all into temporary \texttt{SV}s before pushing them onto the stack:

```c
void preprocess_callout(char** message) {
    dSP;  
    int i;
    ENTER;  
    SAVETMPS;
    PUSHMARK(SP);  
    for (i=0; message[i]; i++)
        XPUSHs(sv_2mortal(newSVpv(message[i], 0)));
    PUTBACK;  
    call_pv("Hermes::preprocess", G_DISCARD);  
    FREETMPS;
    LEAVE;
}
```
As before, we need a copy of the stack pointer so we know where in memory we're putting our parameters.

We begin by opening a new scope with `ENTER` and setting out a new scope for temporary values with `SAVETMPS`.

We iterate over the array, creating a new `SV` for each line of the message (see “Creating SVs with values,” page 106), making them temporary with `sv_2mortal`, and then pushing them onto the stack. If we knew in advance how many lines we had, we could use `EXTEND(n)` to pre-extend the stack and then use `PUSHS` instead of `XPUSHS`; but the approach we use here keeps things simple.

`PUTBACK` sets Perl's copy of the stack pointer equal to the value of the local variable `SP`. Because we've put things onto the stack, our `SP` will have grown; but our changes to the local variable need to be reflected in the global stack pointer.

Now we can call the function. This time we have some arguments, so we don't use `G_NOARGS`; but we're still discarding the return value.

**Return values**

It's not much good preprocessing a message if you don't get the message back after preprocessing it, so you have to deal with return values. Once again, you'll use many of the same principles you saw with respect to `XS` callbacks. We can modify our example slightly:

```c
char** preprocess_callout(char** message) {
    dSP;
    int i;
    int count;
    char **newmessage;
    ENTER;
    SAVETMPS;
    PUSHMARK(SP);
    for (i=0; message[i]; i++)
        XPUSHs(sv_2mortal(newSVpv(message[i], 0)));
    PUTBACK;
    count = call_pv("Hermes::preprocess", G_ARRAY);  // 1
    SPAGAIN;
    newmessage = malloc((count + 1) * sizeof(char*));
    newmessage[count] = NULL;  // 2
    i = count;
    while (i > 0)
        newmessage[--i] = savepv(SvPV_nolen(POPs));  // 3
```
This time we use `G_ARRAY` to specify that we’re calling the function in array context. `call_pv` returns the number of values returned by the subroutine.

Because the subroutine will have put more values on the stack, we need to refresh our local copy of the stack pointer—this is what `SPAGAIN` does. Again, this code is the same as in section 6.7.1.

The way we fetch the values is slightly tricky. First, the values come off the stack in reverse order, so we put the first value to be popped at the end of the array. Second, the values on the stack are temporaries and will be swept away at the `FREETMPS`. Because we don’t want to end up with an array of invalid pointers, we make a copy of each string with `savepv`.

We also need to remember that arrays are zero-based, so if `count = 2`, we should store the first value in `newmessage[1]`. This is why we say `newmessage[--i]`.

### 8.6.2 Context

You’ve seen that you can use the `G_...` flags to affect the context of a call. For instance, you can force scalar context on a list function by passing the `G_SCALAR` flag. The `perlcall` documentation has a comprehensive list of what all the flag values do and how they affect the context of your call.

### 8.6.3 Trapping errors with eval

You may have noticed that if a fatal error is generated by the Perl subroutine, then the entire process gets shut down. You should guard against this occurrence in embedded situations, so you need to be aware of another `G_...` flag.

`G_EVAL` is the equivalent of wrapping the code to be executed in an `eval` `{ ... }` block. Let’s modify our example again to make sure it doesn’t die:

```c
char** preprocess_callout(char** message) {
    dSP;
    int i;
    int count;
    char **newmessage;
    ENTER;
    SAVETMPS;
    PUSHMARK (SP);
```
for (i=0; message[i]; i++)
    XPUSHs(sv_2mortal(newSVpv(message[i], 0)));

PUTBACK;

count = call_pv("Hermes::preprocess", G_ARRAY | G_EVAL); ①
SPAGAIN;

if (SvTRUE(ERRSV)) {
    display_message(
        "An error occurred in the Perl preprocessor: ts",
        SvPV_nolen(ERRSV));
    return message; /* Go with the original */
}

newmessage = malloc((count + 1) * sizeof(char*));
newmessage[count] = NULL;
i = count;

while (i > 0) {
    SV* line_sv = POPs;
    newmessage[--i] = savepv(SvPV_nolen(line_sv));
}

FREETMPS;
LEAVE;
return newmessage;

•

① Adding the G_EVAL flag is all we need to do to protect ourselves from a die in the Perl code.

② Once we've called the subroutine and restored the stack to normality, we check to see whether the error SV ($@) has a true value.

③ Assuming we have a function for displaying formatted messages, we spit out the text of the error message and return the original array unmodified.

8.6.4 Calling Perl methods in C

As you know, there are two types of methods: object methods and class methods. They are both called using the call_method function from C code; the trick is that the object (in the case of an object method) or an SV representing the class name (in the case of a class method) must be placed on the stack before any parameters. Here's an example of calling an object method from C:

PUSHMARK(sp);
XPUSHs(object);
XPUSHs(param);
PUTBACK;
call_method("display", G_DISCARD);
8.6.5 Calling Perl statements

But isn’t this approach a lot of work, just to run some Perl code? There should be a much easier way to run Perl code from C. Thankfully, Perl provides functions called eval_pv and eval_sv, which are essentially the equivalent of eval "...". As with eval, you can do anything you can normally do in a Perl program, including load other modules.

For instance, we can use Perl to nicely format a C string, using the Text::Autoformat module:

```c
char* tidy_string (char* input) {
    SV* result;
    setSVpv(DEFSV, input, 0);
    result = eval_pv("use Text::Autoformat; autoformat($_)", FALSE);
    if (SvTRUE(ERRSV))
        return NULL;
    else
        return SvPV_nolen(result);
}
```

Notice that we store the input string in $_ (DEFSV) and that the second argument to eval_pv and eval_sv is a boolean denoting the behavior on error—if it is TRUE, then the process will exit if the Perl code dies.

In many cases, eval_sv and eval_pv are all you need to get a lot out of embedding Perl in your application. The perlembed man page contains a very good example of using Perl’s regular expression functions from C with these functions.

8.7 Using C in Perl in C

In many cases when embedding Perl in C, you’re providing the user with an alternative way of scripting the program’s behavior. As a result, you often want to provide a way for the user’s Perl to perform actions back in the original C program. For instance, the mod_perl Apache module allows Perl programs to control incoming HTTP requests; this control involves such things as finding out information about the request and sending an HTTP response back to the remote client via the Apache server. These things can’t be done from Perl, and they must be implemented as calls to C functions inside Apache. Thus, in addition to an embedding framework, you need some XS modules to expose the C API of your programs back to the Perl scripts.

Let’s assume we’ve written an XS module to do this. If we try the obvious solution

```c
eval_pv("use Hermes;", TRUE);
```

then this happens:

```
Can’t load module Hermes, dynamic loading not available in this perl.
(You may need to build a new perl executable which either supports
dynamic loading or has the Hermes module statically linked into it.)
```
The very basic Perl interpreter we created does not have support for dynamic loading of XS modules.\(^2\)

Adding this support is easy, especially because ExtUtils::Embed can help us out again. The easiest way to get started is to use ExtUtils::Embed’s xsinit function:

```
% perl -MExtUtils::Embed -e xsinit -- Hermes
```

This command creates a file called `perlxsi.c`, which we can link into our program; it provides a function called `xs_init` that we can pass to `perl_parse`:

```c
int main(int argc, char **argv)
{
    char* command_line[] = {"", "-e",
                           "print "Hello from C!\n";"};

    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, xs_init, 3, command_line, (char **)NULL);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

If you do this, you will also have to link the `Hermes.so` file generated by your XS module to your embedded executable.

A more flexible way to use XS modules is to allow dynamic linking by having `xs_init` load DynaLoader for you. You can do so by not passing any parameters to `xsinit`:

```
% perl -MExtUtils::Embed -e xsinit
```

Now your embedded code will be able to load any Perl module.

### 8.8 **Embedding Wisdom**

Here are a few random pieces of advice for applications that embed Perl. These ideas have generally been discovered through bitter experience but left undocumented, so we’ve gathered them together to save you a little time in your own projects:

- Never call `perl_parse` more than once; you’ll only leak memory.
- Setting the global variable `PL_destruct_level` to 1 may help if you’re having problems with values not being freed properly.
- Avoid using Perl API macros as arguments to other Perl API macros (this advice is also relevant for XS programming).

\(^2\) This is the difference between `miniperlmain.c`, from which we took our first example, and `perlmain.c`, which is the main file used in building an ordinary `perl`. 
8.9 **SUMMARY**

Embedding a Perl interpreter into an application is a great way to increase its flexibility, extensibility, and scriptability. Nevertheless, it comes at a price, in terms of possible memory leaks, a potential drop in speed, and the added complexity of coding the embedding.

We've shown you how to add a Perl interpreter to a program and how to generate build information for the resulting program; we've built a simple interpreter using the `alloc/construct/parse/run/destruct/free` cycle. Once you have an interpreter, you can get at Perl values using `get_sv` and also use any of the functions from the Perl API to communicate with Perl; these abilities include running Perl subroutines via the callback mechanism and examining the Perl stack.

We've also discussed using `eval_pv` as an easy way to execute Perl code and shown you the tricks necessary to allow an embedded interpreter to load XS modules. Finally, we’ve presented some collected wisdom on the implementation of an embedded application.