Embed Perl scripting in C applications

How to add a scriptable element quickly and painlessly

Skill Level: Advanced

Martin C. Brown
Writer and consultant
Freelance

15 Dec 2004

Have you ever wanted a quick way to add a scriptable element to your applications? Embedding an existing language into your application is the simplest and most effective way to incorporate such a system. You get the benefits of an established language to expand the functionality of your application in a flexible way without users having to rebuild the application to use it. This tutorial shows you how to embed a scripting language into an application. You see how to build the application and how to provide wrapper functions with full argument and return value support.

Section 1. Before you start

About this tutorial

Have you ever encountered an application or system that required access to a scriptable environment or component? One obvious reason for such a requirement is to make use of functionality provided by a scripting language that would be difficult or complex to achieve with C. The Perl language is an ideal candidate for tasks such as text processing, parsing, and reformatting. Perl is a practical and easy-to-use scripting language that has the advantage of a rich and well-designed application program interface (API) that enables the language to integrate with host applications.

This tutorial covers:

- How Perl works
- Perl data types
- Embedding the interpreter
- Running Perl scripts within your application
- Exposing your application to the Perl scripts

Prerequisites

This tutorial is written for developers who want to know how to incorporate the Perl interpreter into applications and how to expose these applications to allow Perl to control and manage the components. To use the techniques in this tutorial, you must be proficient in C and have at least a basic knowledge of Perl, its structure, and how to write basic applications in the Perl language.

To run the code samples in this tutorial, you'll need:

- Perl
- A C compiler, such as GCC

Section 2. Introduction to embedding Perl scripting

Why use embedding?

The key reason to embed Perl scripting in your application is to make use of functionality that would be difficult to reproduce in C without significant coding. Classic examples include the text processing and handling that Perl offers -- in particular, the ability to parse text and extract specific elements or merely to use the regular expression engine.

I've used embedded Perl in the past for all manner of tasks -- even in place of the genuine C library for communicating over a network -- because Perl can be easier to use for certain operations. I've also used Perl in more extensive installations in which the Perl code has been an integral part of the application for executing certain fragments of code. It probably comes as no surprise that Perl itself is written in C,

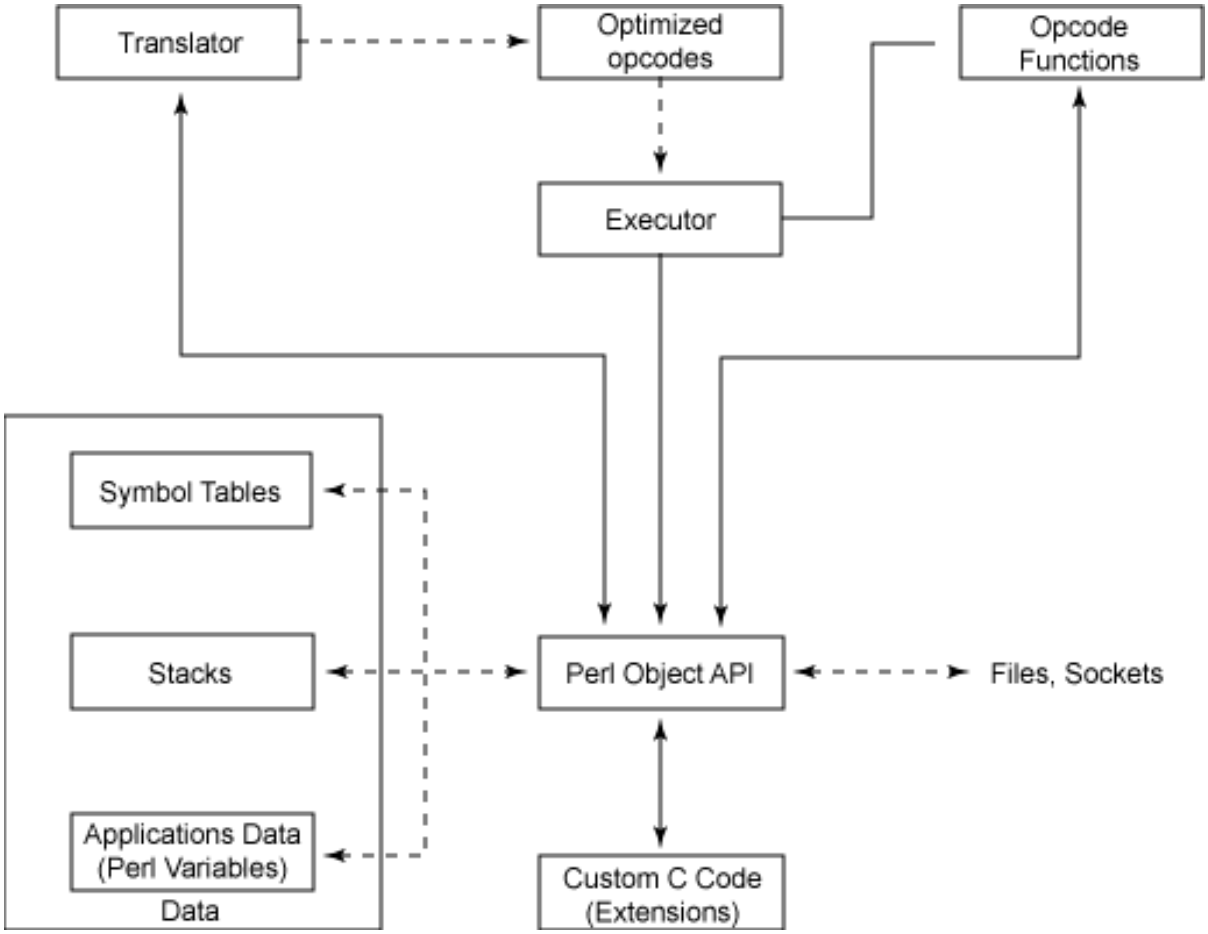
and the Perl interpreter includes an extensive API. Even the `perl` executable uses the Perl API to create a Perl interpreter and execute the script.

The first step in understanding how to embed Perl into your applications is to understand how Perl works internally. Let's take a look at how Perl interprets your Perl scripts.

Inside Perl

Looking at how Perl works can show you how the different components required during a typical embedding process can be used together to support an internal scripting language for your application. Figure 1 shows a basic structural diagram for the Perl language.

Figure 1. Internal Perl architecture



Perl is a scripted language, but as part of the interpretation process, the Perl script that you supply is converted into a series of opcodes. *Opcodes* are small fragments of script that make up one or more sequences in your application. Opcodes are technically similar to the individual instructions executed on a typical CPU. For

example, Perl would translate the expression `$a = $b+$c` by using the "add" opcode to convert it to an opcode structure. The basic Perl instruction set contains about 350 opcodes that handle everything from the basic mathematical functions to the more complex Perl internal functions like `grep`. Other opcodes interface to external functions and C libraries through the Perl extension interface. All opcodes are essentially C functions: Some have been optimized in C to produce the best performance, while others are simply interfaces to the corresponding function in the C library.

When you execute a Perl script, the structure of the application is turned into an opcode sequence. The opcodes, or more specifically the functions that support the individual opcodes, are then executed by the Perl virtual machine. These opcodes work with specific data structures that give Perl its power with the built-in data types of scalars, arrays, and hashes.

Because these opcodes and the structures and functions used to support them are written in C, you can interface with a Perl interpreter within your applications by creating the necessary data structures, then executing Perl scripts using the Perl virtual machine to manipulate the information. This interface is largely one way: you can populate variables and write scripts. For full integration, you need to expose the functions to Perl within your application by writing an interface that translates between the data type you use in C and the data structures used in Perl. Let's start by looking at each main data type and how you can create, update, and manipulate these values within C.

Section 3. Perl data types

Perl's internal types

For the execution unit to work, it has to have set values and data structures to work with. The opcodes are the executable components of the system, and they work on several different *values*. Value types exist for:

- **Scalar values:** Perl stores integers, floats, strings, and references (among other types) in the internal scalar value type. Within the C interface to the Perl internals, separate functions exist for creating different scalar types based on the corresponding data types that C employs.
- **Array values:** Array values are essentially just arrays of the internal

scalar value type. Unlike standard C array types, Perl arrays can be automatically extended; the Perl internals will work out the memory allocation and other problems for you.

- **Hash values:** Hashes have no direct equivalent in C, but the functions provided enable you to create and control the contents of the hashes that you create.
- **Glob values:** Glob values are used for almost every other type of information. They are responsible for holding references to filehandles and formats as well as providing a "catch-all" solution for accessing references to other data types.
- **Code values:** Code values are not often referred to as such; they make up the executable code blocks, functions, and components. Unlike the other value types, code values are usually accessed by directly executing a code block or function by name rather than by referring to the code values directly.

To work with Perl through the C interface, you must work with, create, and manipulate the information stored in these different value types. Let's look more closely at how to create and use these different types.

Creating scalar values

There are common themes to the entire range of functions for manipulating Perl values, largely because these values are based on a common theme. For the array and hash values, the subtypes are scalar values; you'll use most of the functions that help you manipulate scalar values when you're manipulating array contents.

Because Perl types don't match internal C types, you have to use functions to create, update, and store information in Perl structures. When you've created the variables in the Perl structure, the variables can be used in and even made accessible to a Perl script embedded within an application.

To create a new scalar value, you need to use one of the following functions. Each function returns a pointer to a value (SV *):

- **`newSViv(long i)`** creates a new scalar value from the supplied integer type `i`. This function creates an IV (Integer Value) type SV.
- **`newSVnv(double f)`** creates a new scalar value from the supplied float or double `f`. This function creates an NV (Numerical (double) Value) type SV.
- **`newSVpv(char str, int len)`** creates a new scalar value from the supplied string `str`, of length `len`. If `len` is zero, Perl will calculate the

string length for you. This function creates a PV (string value) type SV.

Note that you can't assign any value to create a new scalar value. Instead, you have to tell the Perl interface which type of scalar value you're creating, which goes against the Perl ethos that a scalar value can contain anything. You need to be aware of this issue while building an interface to the internals, because you must use the right function in time.

Manipulating scalar values

In scripts, Perl treats scalar values as a catch-all structure; but internally, the information is stored separately. Within a Perl script, for example, it's normal to assign a number to a scalar value, then incorporate that number into a string to print it out. The reverse also happens: You assign a string to Perl that just happens to be a number. Initially, Perl will store this value as a string, but it converts the value to a number if you use it in a numerical expression. Behind the scenes, if you create a scalar value in a Perl script and assign an integer to it, that is how the value is stored internally within the corresponding SV. The value is converted into a float or a string value only if your Perl script requires it as such, and Perl does the translation automatically. No such benefit exists within C. If you want to perform similar tasks in your C interface, you must do so manually.

Three sets of functions are useful for manipulating scalar values. The first set returns the type of value stored in a scalar value. The second set changes an existing scalar value into a new value of a specific type. The third set converts a scalar value so that the stored value is identified as a specific type. First, let's look at the type functions:

- **`SvIOK(SV *)`** returns true if the scalar value is an integer value.
- **`SvNOK(SV *)`** returns true if the scalar value is a double value.
- **`SvPOK(SV *)`** returns true if the scalar value is a string value.

Additional functions exist for determining whether a scalar value is a reference, an undefined, or a "true" value.

You can also use the `SVTYPE(SV *)` function, which will return a value specifying the type. Several macros exist for the various types: `SVt_IV` (integer), `SVt_NV` (double), and `SVt_PV` (string). Other macros exist for references and for references to specific types (array, hash, code, glob).

To change an existing scalar value, you use one of the `sv_set*` functions. For example, the following code fragment would initialize an integer SV, then assign a double to it:

```
SV * myscalar;
myscalar = newSViv(3); /* myscalar is type SVt_IV */
sv_setnv(myscalar,3.14159265); /* myscalar is now SVt_NV */
```

To change the "default" type of a scalar, you use `SvIV(SV *)` to convert to an integer, `SvNV(SV *)` to convert to a double, or `SvPV(SV *, int len)` to convert to a string. When converting a double to an integer, the fractional component is removed. When converting a string to either an IV or an NV, Perl uses the equivalent of `sscanf()` to convert into the corresponding value.

Freeing and deleting scalar values

Many more functions are available to help you set, use, and work with scalar values within Perl. For example, an entire set of instructions exists for working with references to other structures and for working with scalar values that are objects or parts of other packages. These components aren't applicable to this tutorial, but check out the `perlapi` man page if you want more information.

A couple of other functions that are useful are `perl_get_sv(char *varname, int create)` and `sv_dump(SV *)`. The `perl_get_sv(char *varname, int create)` function converts the scalar value named within a Perl script into a C SV structure. For example, if you run a script like this:

```
$returnstring = my_perl_function()
```

you can obtain a pointer to that scalar value by calling

```
SV *returnstring = perl_get_sv('main::returnstring',0)
```

If the `create` argument is not zero, the function will create a scalar variable of that name.

The other useful function, `sv_dump(SV *)`, "pretty-prints" a scalar value, irrespective of its built-in type. This function can be a useful debugging tool during development that allows you to check the value of different components.

With the functions we've covered so far, it's up to you to free and delete any scalar values that you create. This task can be time consuming, especially if you create and use scalar values a lot, and you'll need to code very carefully and make sure you release the scalars after use. Alternatively, you can let Perl handle the task for you by marking the scalar value as *mortal*. Doing so will automatically free up the memory when the variable goes out of scope (for example, within a function). You can mark an existing scalar value as mortal by calling `sv_2mortal(SV *)`, or you can create a "blank" mortal scalar value by using `sv_newmortal(SV *)`.

Manipulating arrays

Now that you understand the basics of scalar values, manipulating arrays is comparatively straightforward. Because an array value is a special type, you can't just use an array as you would within C: All access -- from adding elements to an array through extracting values -- is through functions. Remember at all times that array values store pointers to scalar values, so the updating and extraction process uses scalar values at each stage.

You can create an array in either of two ways: You can create an empty array value to be updated later, or you can create a populated array value by supplying a pointer to an array of scalar values. You use the `AV *newAV()` function to create blank array values. To create a populated array value, you use the `AV *av_make(int num, SV **ptr)` function.

You determine the array value's length through the `av_len(AV *)` function; you fetch a scalar value by using

```
av_fetch(AV *, I32 index, I32 lval)
```

which returns the SV at the `index` within the array. The `lval` should remain zero if you want only to read the information or non-zero if you want the value set to `undef`. To store values in the array, you use

```
av_store(AV *, I32 index, SV *val)
```

Note that this is the only way to get values directly into the array at specific locations.

Finally, the most common manipulations to arrays -- inside and outside Perl scripts -- is through adding or removing items. Within Perl, this manipulation is handled through the `push`, `pop`, `shift`, and `unshift` functions. These functions work identically in the C API to Perl as they do within Perl itself with one exception: the `av_unshift(AV *, I32 num)` function. Unlike the Perl function, this function adds only `num` elements to the array, and you'll need to use the `av_store` function to actually update the elements with a value.

Manipulating hashes

If you can handle an array, handling a hash is not that difficult. Like arrays, hashes store scalar values. Keys may be specified either as a string or as a scalar value, although you need to use different functions accordingly. Hashes can be easy to work with because you can get away with a string from the C interact.

You can create a new hash value by using the `newHV()` function. To store a value, using a character string as the key, use

```
hv_store(HV *hash, char *key, U32 klen, SV *val, U32 hashval)
```

To get a scalar value back, use

```
SV **hv_fetch(HV *hash, char *key, U32 klen, I32 lval)
```

You can also iterate through key/value pairs in the hash by using a combination of two functions. The `hv_iterinit(HV *hash)` function initializes the iteration, returning the number of elements in the hash in the process. You then iterate through each pair using

```
SV *hv_iternextsv(HV *hash, char **key, I32 *pkeylen)
```

The `key` is updated with the key, and the return value is the corresponding value in the hash.

Finally, you can obtain the hash value of an existing hash by using

```
HV *perl_get_hv(char *varname, int create)
```

if you set the `create` argument to `0`. (If you set the `create` argument to `1`, you will create a hash with the supplied name.) Note that the name must be fully qualified within the scope of a typical Perl interpreter (that is, within a specific package, such as "main").

Now that you know the basics of the interaction between variables and values from within a C interface, let's look at how to embed the interpreter, then use these functions to manipulate the information and values generated in the process.

Section 4. Basic embedding

Creating a simple Perl wrapper

Simply embedding the Perl interpreter is an easy task. Use this code to create a new wrapper (that is, a simple C-based application) around the Perl interpreter:

```
#include <EXTERN.h>
#include <perl.h>
static PerlInterpreter *my_perl;
int main(int argc, char **argv, char **env)
{
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, argc, argv,
env);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

The two header files are required for the rest of the code. The `EXTERN.h` contains the definitions necessary for integrating with external components for use with Perl. The `PerlInterpreter` type is a structure that holds all the information for a single instance of the Perl interpreter. It's possible to have more than one Perl interpreter active at any one time, although in practice, you rarely have more than one in a single application.

Within the `main` function, the `perl_alloc()` function allocates the resources for an instance of the Perl interpreter, and `perl_construct()` creates the instance. The `perl_parse()` function parses Perl code. Note that you supply the Perl interpreter with both the arguments and the environment from the host program. (You could just as easily have created your own specific environment for the interpreter.) The second argument can contain any initialization routines you might need; in most instances, you use the second argument to initialize the XS interface to C-based Perl extension modules, although it can also be used for any Perl-specific initializations.

When the Perl table is ready and initiated, you call the `perl_run()` function to execute the instance of the interpreter. Because you haven't supplied the interpreter with any actual code to execute, the interpreter will ask for it from stdin when the application is executed. The cleanup isn't strictly necessary in this case, because when the program exits, the instance will be closed and the memory freed as part of normal process destruction. However, it's a good habit to get into.

Compiling an embedded Perl application

Perl comes in many different versions and can be installed in just about any location. Providing a command line that would compile the above application and would be applicable for all platforms and environments would be impossible if it weren't for the information that the `ExtUtils::Embed` module provides. This module can generate the necessary C compiler and linker options, irrespective of how Perl was installed or configured.

To compile the application above, you use a command such as:

```
$ cc -o myperl myperl.c `perl -MExtUtils::Embed -e ccopts -e
ldopts`
```

Of course, the exact output generated depends on your system and environment.

If you run the newly compiled program, you'll find that you've produced your own version of the wrapper around the Perl interpreter:

```
 $ myperl
print join(', ',keys %ENV),"\n";
EOF
PWD, LOGNAME, SHLVL, BASH_ENV, XSHELL, SHELL, DYLD_LIBRARY_PATH,
USER, SSH_TTY, CC, VISUAL, TMPDIR, DISTCC_HOSTS, HISTSIZE,
LS_COLORS, USERNAME, SSH_CLIENT, LESSOPEN, EDITOR, ASAVE,
TERM, SSH_CONNECTION, CCACHE_PREFIX, CCACHE_LOGFILE, CVSROOT,
MAIL, LANG, CLASSPATH, PATH, HOSTNAME, LD_LIBRARY_PATH,
SSH_ASKPASS, MANPATH, MAILCHECK, HOME, ROGUEOPTS, PS1,
cw, JAVA_HOME, QTDIR, _, LANGTYPE, CCACHE_DIR, TEXINPUTS,
INPUTRC, G_BROKEN_FILENAMES
```

You've now executed your first bit of Perl code from your C program. However, having to type in the Perl you want to run is less than ideal, not to mention that there isn't a great deal of communication between the C wrapper and the Perl program that you generated.

Calling extensions from embedded interpreters

To use extension modules (that is, any module that relies on a C interface to an external library) within an embedded Perl interpreter, you must change the way in which the Perl instance is initialized; otherwise, Perl won't be able to load the external components. This initialization function is then supplied to the `perl_parse` call:

```
perl_parse(my_perl, xs_init, argc, argv, env);
```

The `xs_init` function is relatively straightforward, but don't bother creating it yourself. Instead, use the `ExtUtils::Embed` interface to generate it for you:

```
$ perl -MExtUtils::Embed -e xsinit - -o xsinit.c
$ cc -c xsinit.c `perl -MExtUtils::Embed -e ccopts`
```

Then, when building and linking your embedded application, supply the compiled `xsinit.o` as an additional source object:

```
$ cc -c myperl.c `perl -MExtUtils::Embed -e ccopts`
$ cc -o myperl myperl.o xsinit.o `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

If you want to simplify the build process, you can put all this code into a Makefile.

Section 5. Integrating with a script

Calling a function

At this point, all you've really done is reproduce the Perl interpreter. But the interpreter doesn't do anything until you run the application and supply it with Perl code; the real utility comes from being able to run arbitrary Perl fragments and use information from the enclosing C application within those scripts.

To call a specific Perl function, use the `call_argv` function. This function calls a named function, including the ability to supply any arguments to that Perl function. Here's an example of an application to call the print function:

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

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
  char *print_args[] =
{"This","is","a","list","of","printable","items","\n",NULL};

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

  call_argv("printwrap", G_DISCARD, print_args);

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

The key part of this application is the creation of a list of arguments, which you supply as a pointer to a list of character strings terminated with a single NULL element, and the call to `call_argv` itself. Note that you can't call `print` directly; instead, you call a wrapper function, as you see here:

```
sub printwrap
{
    my (@words) = @_;
    print @words;
}
```

Compile the code as `myperlfunc`, then save the Perl script as `printwrap.pl`. Use this command to execute the program:

```
$ myperlfunc printwrap.pl
Thisisalistofprintableitems
```

The `call_argv()` function can be useful in those situations in which you want to execute arbitrary Perl code within your application and generate a result -- all entirely from within Perl. You can pass arguments to the functions as necessary (as you see in the script above). However, to get return values from a call, you need to use a different method.

The Perl stack

So far, you haven't been able to use Perl to execute a function and get a return value for use in your application. Doing so is the next step in understanding and using Perl in an embedded situation. To use Perl in this way, you must integrate with the stack; the `call_argv` function does this for you automatically when providing values to a function. But to get information back, you need to manipulate the stack directly.

The stack in Perl is used to exchange data between functions. When you call a function, any arguments you put into the function call are in fact pushed onto the stack. The function within Perl then takes the arguments back off the stack to process them. A *stack* is essentially just a big array, and the stack is populated and depopulated in much the same way as an array that you control through the Perl `push` and `pop` functions.

In real terms, there are actually two stacks: an *argument stack,* which contains the actual arguments, and a *markstack,* which contains the pointers to the argument stack. The combination of these two stacks prevents you from taking too many variables from the stack within a function. The `@_` variable is populated only with the variables on the argument stack up to the limit of the latest pointer on the markstack. You can see the relationship more clearly in Figure 2, which shows the state of the two stacks when calling a simple two-argument `add(a,b)` function within Perl.

Figure 2. The Perl stack

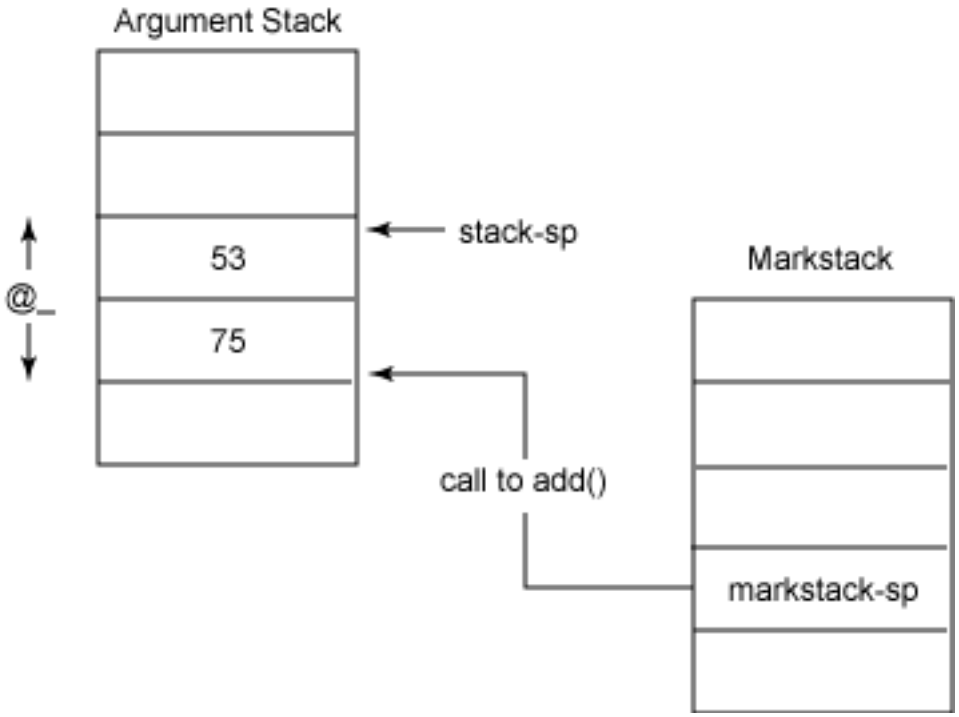


Figure 2 shows the state of the two stacks at the instant of the function call. There are two arguments on the stack, and the `markstack-sp` (markstack stack pointer) points to the start of elements placed onto the argument stack for the `add()` function call. Meanwhile, the `stack-sp` (argument stack stack pointer) points to the end of the entire argument stack.

Manipulating the stack

Stack manipulation is based on another suite of functions. Here you see an example of a sequence; the functions used at each stage appear in parentheses:

1. Initialize the environment (dSP).
2. Start the scope (ENTER).
3. Set up the environment so that temporary variables will be automatically deleted at the end of the scope (SAVETMPS).
4. Remember the current top of the stack (PUSHMARK).
5. Push arguments onto the stack (XPUSH*).
6. Signify the end of the arguments (PUTBACK).

7. Call the Perl function.
8. Signify the start of return value reclamation (SPAGAIN).
9. Get the values from the stack (POP*).
10. Signify the end of return value reclamation (PUTBACK).
11. Free the temporary variables.
12. End the scope (LEAVE).

Now, let's put this sequence into practice.

Creating a two-way function call

To understand this sequence, I'll show you how to build a wrapper around the Perl code you see below. This code takes a string and a separator and returns a list of the separated words in reverse order. I've added two statements to help print out the information I'm working with, just for reference.

```
sub reverse
{
    my ($string,$separator) = @_;
    my @words = split /$separator/,$string;
    print "Words in source are: ",join(", ", @words),"\n";
    my @sorted = sort { lc($b) cmp lc($a) } @words;
    print "Words in return are: ",join(", ", @sorted),"\n";
    return @sorted;
}
```

To use this code within C, you need to build a wrapper function that will populate the stack, call the function, then read the return values back again. You can then call this wrapper function from within a C application to run the Perl code. Because you've built the wrapper, it will look to your C application like any other function call.

Creating a two-way C wrapper

For the wrapper function, the Perl fragment follows the basic sequence you just saw: initializing your connection and manipulation of the stack, pushing the arguments to the Perl function onto the stack, calling the Perl function, then taking the return

values off again. Here's the code:

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

static PerlInterpreter *my_perl;

void perl_reverse(char *string, char *separator)
{
  int retval;
  STRLEN n_a;

  dSP;
  ENTER;
  SAVETMPS;
  PUSHMARK(sp);
  XPUSHs(sv_2mortal(newSVpv(string,0)));
  XPUSHs(sv_2mortal(newSVpv(separator,0)));
  PUTBACK;
  retval = perl_call_pv("reverse", G_ARRAY);
  SPAGAIN;
  if (retval > 0)
    while(retval-- != 0)
      printf("Returned: %s\n",POPpx);
  PUTBACK;
  FREETMPS;
  LEAVE;
}

int main(int argc, char **argv, char **env)
{
  char *my_argv[] = { "", "reverse.pl" };
  my_perl = perl_alloc();
  perl_construct(my_perl);
  perl_parse(my_perl, NULL, argc, my_argv, NULL);

  perl_reverse("Come grow old along with me"," ");

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

Look at the `perl_reverse` function first. The basic sequence is as previously described. The important portions are the calls to `XPUSHs`, which pushes a string onto the stack. The `XPUSH*` functions increase the stack size, then push the value onto it. Note that previously, you used the `SAVETMPS` function: This function will free up any temporary mortal scalar values created during this process when you call `FREETMPS`. Therefore, you create a mortal scalar value based on the SV string type. The call to the function is embedded in the call to `call_pv`. This function returns the number of arguments placed onto the stack when the function returned.

Getting return values from the stack

The final stage consists of reading each argument from the stack and, in this case, printing out the word that was returned. The `POPpx` function returns a string (you force Perl not to care what the length might be by defining the `STRLEN n_a`). You

simply decrement the return value until you've read everything from the stack. Note, however, that you've deliberately read a string (actually, a pointer to a string) off the stack. It's generally best to write a wrapper that returns a known value type so that you can specifically remove a value of that type from the stack each time. Also note that you've popped a C type off the stack, not an SV.

The `main` section has just two changes. You still initialize and free the Perl interpreter, but the call to the Perl interpreter is handled by your wrapper function. The other change is that unlike previous examples, you populate your own `argv`, which tells Perl to load the script with the function definition automatically without your having to specify the script at the command line. Doing so simplifies execution of the application to:

```
$ ./reverse
Words in source are: Come, grow, old, along, with, me
Words in return are: with, old, me, grow, Come, along
Returned: along
Returned: Come
Returned: grow
Returned: me
Returned: old
Returned: with
```

You will have noticed that the output isn't in the reverse order you expected. That's because when Perl puts values onto the stack, they are put on in the order supplied, but the `POP*` function takes them off in reverse order. To get them back in the order they were put on the stack, you need to alter the code to take account of this order:

```
if (retval > 0)
{
  counter = retval;
  while(counter-- != 0)
    strings[counter] = POPpx;
  while(counter++ < (retval-1))
    printf("Returned: %s\n",strings[counter]);
}
```

If you're dealing with values other than strings, you'll need to account for the inverse nature of the system to ensure that you get back the right return values into the right variables.

It's worth noting that when using the `XPUSH*` function, you're relying on Perl to manage the stack for you, including extending and managing the stack to ensure that sufficient space exists to push values onto it. While this is fine for relatively small numbers of arguments and return values, you can experience problems with large stack sizes. I've experienced problems with manipulations as small as 10 items, but you can imagine that with some array-based functions, the problems become unwieldy.

Such a problem isn't a function of the C API: the same can be true within Perl, although the effects are seen less often because Perl has complete control over the environment in which the function executes. In both systems, though, performance can be an issue. Each PUSH onto the stack may mean increasing the memory allocation plus adding the element itself. For hundreds or even thousands of elements, that becomes a major problem. In Perl, the obvious solution is to use a reference to an array rather than exchanging the array itself. You can do the same from the C API by using the techniques in the next section.

Section 6. Using references to simplify stack interaction

Using references within Perl

One solution to the problem of inconsistencies between the supplied arguments and the return values is to use references to hashes or arrays. Using this method, you can supply arguments and obtain return values without pushing more than one item onto the stack in either direction. This tactic is fairly common in Perl, and you can use it here.

First, create a Perl script that defines the function you'll use. In essence, this script is the same as the last script, except that now you extract the information to be processed from a hash and supply both a forward- and a reverse-sorted version of the word list as a hash reference on return:

```
sub reorder
{
    my ($args) = @_;

    my $string = $args->{string};
    my $separator = $args->{separator};

    my @words = split /$separator/,$string;

    print "Words in source are: ",join(", ", @words),"\n";

    my $forward = join (', ',sort { lc($b) cmp lc($a) }
@words);
    my $reverse = join (', ',sort { lc($a) cmp lc($b) }
@words);

    my $retvalues = {'forward' => $forward,
                     'reverse' => $reverse,};

    return $retvalues;
}
```

Using references requires a little more pre- and post-processing in the function, but in essence, the content is no different. That said, however, the C wrapper function is more complicated.

Using references in the C wrapper

The C code for this script is more complex. This time, you have to populate a hash with the arguments you want to supply, then create a reference to this hash and push that reference onto the stack. You do more or less the reverse when the information has come back: You have to pop the reference to the returned hash off the stack, dereference the hash, iterate through the hash's key/value pairs, and print out the results. Here's the full code that you'll use:

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

static PerlInterpreter *my_perl;

void perl_reorder(char *string, char *separator)
{
  int retval;
  HV * arghash;
  HV * rethash;
  SV * hashref;
  char *key;
  SV *value;
  int keycount;
  I32 *keylen;
  STRLEN n_a;

  arghash = newHV();
  rethash = newHV();

  hv_store(arghash, "string", 6, sv_2mortal(newSVpv(string,0)),0);
  hv_store(arghash, "separator", 9, sv_2mortal(newSVpv(separator,0)),0);

  hashref = newRV_inc((SV *)arghash);

  dSP;
  ENTER;
  SAVETMPS;
  PUSHMARK(sp);
  XPUSHs(sv_2mortal(hashref));
  PUTBACK;
  retval = perl_call_pv("reorder", G_ARRAY);
  SPAGAIN;
  if (retval == 1)
    {
      rethash = (HV *)SvRV(POPs);
      keycount = hv_iterinit(rethash);
      printf("%d items in return\n",keycount);
      while(keycount-- != 0)
        {
          value = hv_iternextsv(rethash, &key, &keylen);
          printf("Return (%s) -> (%s)\n",key,SvPV_nolen(value));
        }
```

```
    }

  PUTBACK;
  FREETMPS;
  LEAVE;
}

int main(int argc, char **argv, char **env)
{
  char *my_argv[] = { "", "reversehash.pl" };
  my_perl = perl_alloc();
  perl_construct(my_perl);
  perl_parse(my_perl, NULL, argc, my_argv, NULL);

  perl_reorder("Come grow old along with me"," ");

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

Let's take a closer look at the key components of the wrapper.

The C wrapper

The vital segment is:

```
  hv_store(arghash, "string", 6, sv_2mortal(newSVpv(string,0)),0);
  hv_store(arghash, "separator", 9, sv_2mortal(newSVpv(separator,0)),0);

  hashref = newRV_inc((SV *)arghash);
```

This segment of code populates a hash with the arguments supplied to this function, creating mortal scalar values in the process. Note that you're using bare strings as keys. The final process is to create a reference to the hash so that you can push it onto the stack.

Equally important is the segment of code that prints the results:

```
      rethash = (HV *)SvRV(POPs);
      keycount = hv_iterinit(rethash);
      printf("%d items in return\n",keycount);
      while(keycount-- != 0)
        {
          value = hv_iternextsv(rethash, &key, &keylen);
          printf("Return (%s) -> (%s)\n",key,SvPV_nolen(value));
        }
```

To get the information back, you pop the reference to the hash off the stack (using the `POPs` function), then de-reference it, casting it to a hash value using `SvRV`. Then, it's a simple case of iterating through the key/value pairs and printing out the

information.

You should now be armed with enough information to embed basic Perl scripts into your applications. Obviously, in a larger context, you'd specify a single script that could integrate with several functions in your host application. Whenever you needed to use Perl, you just call one of the wrapper functions that calls the corresponding function.

Section 7. Summary

We've now covered the basics of embedding Perl functions and expressions into a C program. The easiest way to perform such tasks is to use a direct call to the custom function that you want to use. But if you want to deal with return values, you need to work with Perl variables and the Perl stacks to populate the function arguments and extract the values that the function returns. For more complex two-way interactions, you can even use hashes and references to supply and recover information from the function you're calling.

The basic methods shown here can apply to other solutions and interfaces. For example, there is an additional suite of functions for interacting with and calling methods on objects within a Perl script. These methods can be used in combination with Perl extensions to provide access to external components and systems -- even those that have their own C API, like MySQL and Web services.

Although not covered here, you can also use extensions in other ways within an embedded application. Imagine, for example, that you have created an extension to the components in your application. Using an embedded script, you could use Perl to control elements of your application. Game developers use combinations like this to provide scriptable components such as the actors and environments that you interact with. By using an embedded script language, these developers can modify the behavior of different components without having to rebuild them.

Resources

Learn

- Get started with Perl development at the home of Perl.
- For tips on how to squeeze the most performance from your Perl code, see Martin's article "Optimize Perl" (developerWorks, October 2004).
- Simplify the installation of any open source software project with Martin's article "Automate the application build and distribution process" (developerWorks, September 2004).
- The Comprehensive Perl Archive Network is the home of the CPAN network system.
- Find more tutorials for Linux developers in the developerWorks Linux zone.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Download IBM trial software directly from developerWorks.

Discuss

- Read developerWorks blogs, and get involved in the developerWorks community.

About the author

Martin C. Brown
Martin C. Brown is a former IT Director with experience in cross-platform integration. A keen developer, he has produced dynamic sites for blue-chip customers, including HP and Oracle, and is the Technical Director of Foodware.net. Now a freelance writer and consultant, MC (as he is better known) works closely with Microsoft as an SME, is the LAMP Technologies Editor for *LinuxWorld* magazine, is a core member of the AnswerSquad.com team, and has written several books on topics as diverse as Microsoft certification, iMacs, and open source programming. Despite his best attempts, he remains a regular and voracious programmer on many platforms and numerous environments. Contact MC at questions@mcslp.com or through his Web site.