NAME

perlXStut - Tutorial for writing XSUBs

DESCRIPTION

This tutorial will educate the reader on the steps involved in creating a Perl extension. The reader is assumed to have access to *perlguts*, *perlapi* and *perlxs*.

This tutorial starts with very simple examples and becomes more complex, with each new example adding new features. Certain concepts may not be completely explained until later in the tutorial in order to slowly ease the reader into building extensions.

This tutorial was written from a Unix point of view. Where I know them to be otherwise different for other platforms (e.g. Win32), I will list them. If you find something that was missed, please let me know.

SPECIAL NOTES

make

This tutorial assumes that the make program that Perl is configured to use is called `make`. Instead of running "make" in the examples that follow, you may have to substitute whatever make program Perl has been configured to use. Running **perl -V:make** should tell you what it is.

Version caveat

When writing a Perl extension for general consumption, one should expect that the extension will be used with versions of Perl different from the version available on your machine. Since you are reading this document, the version of Perl on your machine is probably 5.005 or later, but the users of your extension may have more ancient versions.

To understand what kinds of incompatibilities one may expect, and in the rare case that the version of Perl on your machine is older than this document, see the section on "Troubleshooting these Examples" for more information.

If your extension uses some features of Perl which are not available on older releases of Perl, your users would appreciate an early meaningful warning. You would probably put this information into the *README* file, but nowadays installation of extensions may be performed automatically, guided by *CPAN.pm* module or other tools.

In MakeMaker-based installations, *Makefile.PL* provides the earliest opportunity to perform version checks. One can put something like this in *Makefile.PL* for this purpose:

```
    eval { require 5.007 }
        or die <<EOD;
    ############
    ### This module uses frobnication framework which is not available
before
    ### version 5.007 of Perl.  Upgrade your Perl before installing
Kara::Mba.
    ############
    EOD
```

Dynamic Loading versus Static Loading

It is commonly thought that if a system does not have the capability to dynamically load a library, you cannot build XSUBs. This is incorrect. You *can* build them, but you must link the XSUBs subroutines with the rest of Perl, creating a new executable. This situation is similar to Perl 4.

This tutorial can still be used on such a system. The XSUB build mechanism will check the system and build a dynamically-loadable library if possible, or else a static library and then, optionally, a new statically-linked executable with that static library linked in.

Should you wish to build a statically-linked executable on a system which can dynamically load libraries, you may, in all the following examples, where the command "`make`" with no arguments is executed, run the command "`make perl`" instead.

If you have generated such a statically-linked executable by choice, then instead of saying "`make test`", you should say "`make test_static`". On systems that cannot build dynamically-loadable libraries at all, simply saying "`make test`" is sufficient.

TUTORIAL

Now let's go on with the show!

EXAMPLE 1

Our first extension will be very simple. When we call the routine in the extension, it will print out a well-known message and return.

Run "`h2xs -A -n Mytest`". This creates a directory named Mytest, possibly under ext/ if that directory exists in the current working directory. Several files will be created in the Mytest dir, including MANIFEST, Makefile.PL, Mytest.pm, Mytest.xs, Mytest.t, and Changes.

The MANIFEST file contains the names of all the files just created in the Mytest directory.

The file Makefile.PL should look something like this:

```
    use ExtUtils::MakeMaker;
    # See lib/ExtUtils/MakeMaker.pm for details of how to influence
    # the contents of the Makefile that is written.
    WriteMakefile(
NAME         => 'Mytest',
VERSION_FROM => 'Mytest.pm', # finds $VERSION
LIBS         => [''],   # e.g., '-lm'
DEFINE       => '',     # e.g., '-DHAVE_SOMETHING'
INC          => '',     # e.g., '-I/usr/include/other'
    );
```

The file Mytest.pm should start with something like this:

```
    package Mytest;

    use 5.008008;
    use strict;
    use warnings;

    require Exporter;

    our @ISA = qw(Exporter);
    our %EXPORT_TAGS = ( 'all' => [ qw(

    ) ] );

    our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );

    our @EXPORT = qw(

    );
```

```
    our $VERSION = '0.01';

    require XSLoader;
    XSLoader::load('Mytest', $VERSION);

    # Preloaded methods go here.

    1;
    __END__
    # Below is the stub of documentation for your module. You better edit
it!
```

The rest of the .pm file contains sample code for providing documentation for the extension.

Finally, the Mytest.xs file should look something like this:

```
    #include "EXTERN.h"
    #include "perl.h"
    #include "XSUB.h"

    #include "ppport.h"

    MODULE = Mytest  PACKAGE = Mytest
```

Let's edit the .xs file by adding this to the end of the file:

```
    void
    hello()
 CODE:
     printf("Hello, world!\n");
```

It is okay for the lines starting at the "CODE:" line to not be indented. However, for readability purposes, it is suggested that you indent CODE: one level and the lines following one more level.

Now we'll run "`perl Makefile.PL`". This will create a real Makefile, which make needs. Its output looks something like:

```
    % perl Makefile.PL
    Checking if your kit is complete...
    Looks good
    Writing Makefile for Mytest
    %
```

Now, running make will produce output that looks something like this (some long lines have been shortened for clarity and some extraneous lines have been deleted):

```
    % make
    cp lib/Mytest.pm blib/lib/Mytest.pm
    perl xsubpp  -typemap typemap  Mytest.xs > Mytest.xsc && mv Mytest.xsc
Mytest.c
    Please specify prototyping behavior for Mytest.xs (see perlxs manual)
    cc -c     Mytest.c
    Running Mkbootstrap for Mytest ()
    chmod 644 Mytest.bs
    rm -f blib/arch/auto/Mytest/Mytest.so
```

```
    cc  -shared -L/usr/local/lib Mytest.o  -o
blib/arch/auto/Mytest/Mytest.so   \
               \

    chmod 755 blib/arch/auto/Mytest/Mytest.so
    cp Mytest.bs blib/arch/auto/Mytest/Mytest.bs
    chmod 644 blib/arch/auto/Mytest/Mytest.bs
    Manifying blib/man3/Mytest.3pm
    %
```

You can safely ignore the line about "prototyping behavior" - it is explained in the section "The PROTOTYPES: Keyword" in *perlxs*.

If you are on a Win32 system, and the build process fails with linker errors for functions in the C library, check if your Perl is configured to use PerlCRT (running **perl -V:libc** should show you if this is the case). If Perl is configured to use PerlCRT, you have to make sure PerlCRT.lib is copied to the same location that msvcrt.lib lives in, so that the compiler can find it on its own. msvcrt.lib is usually found in the Visual C compiler's lib directory (e.g. C:/DevStudio/VC/lib).

Perl has its own special way of easily writing test scripts, but for this example only, we'll create our own test script. Create a file called hello that looks like this:

```
    #! /opt/perl5/bin/perl

    use ExtUtils::testlib;

    use Mytest;

    Mytest::hello();
```

Now we make the script executable (`chmod +x hello`), run the script and we should see the following output:

```
    % ./hello
    Hello, world!
    %
```

EXAMPLE 2

Now let's add to our extension a subroutine that will take a single numeric argument as input and return 0 if the number is even or 1 if the number is odd.

Add the following to the end of Mytest.xs:

```
    int
    is_even(input)
     int input
 CODE:
     RETVAL = (input % 2 == 0);
 OUTPUT:
     RETVAL
```

There does not need to be whitespace at the start of the "`int input`" line, but it is useful for improving readability. Placing a semi-colon at the end of that line is also optional. Any amount and kind of whitespace may be placed between the "`int`" and "`input`".

Now re-run make to rebuild our new shared library.

Now perform the same steps as before, generating a Makefile from the Makefile.PL file, and running make.

In order to test that our extension works, we now need to look at the file Mytest.t. This file is set up to imitate the same kind of testing structure that Perl itself has. Within the test script, you perform a number of tests to confirm the behavior of the extension, printing "ok" when the test is correct, "not ok" when it is not.

```
    use Test::More tests => 4;
    BEGIN { use_ok('Mytest') };

    #########################

    # Insert your test code below, the Test::More module is use()ed here so
 read
    # its man page ( perldoc Test::More ) for help writing this test
script.

    is(&Mytest::is_even(0), 1);
    is(&Mytest::is_even(1), 0);
    is(&Mytest::is_even(2), 1);
```

We will be calling the test script through the command "`make test`". You should see output that looks something like this:

```
    %make test
    PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
    t/Mytest....ok
    All tests successful.
    Files=1, Tests=4,  0 wallclock secs ( 0.03 cusr +  0.00 csys =  0.03
CPU)
    %
```

What has gone on?

The program h2xs is the starting point for creating extensions. In later examples we'll see how we can use h2xs to read header files and generate templates to connect to C routines.

h2xs creates a number of files in the extension directory. The file Makefile.PL is a perl script which will generate a true Makefile to build the extension. We'll take a closer look at it later.

The .pm and .xs files contain the meat of the extension. The .xs file holds the C routines that make up the extension. The .pm file contains routines that tell Perl how to load your extension.

Generating the Makefile and running `make` created a directory called blib (which stands for "build library") in the current working directory. This directory will contain the shared library that we will build. Once we have tested it, we can install it into its final location.

Invoking the test script via "`make test`" did something very important. It invoked perl with all those `-I` arguments so that it could find the various files that are part of the extension. It is *very* important that while you are still testing extensions that you use "`make test`". If you try to run the test script all by itself, you will get a fatal error. Another reason it is important to use "`make test`" to run your test script is that if you are testing an upgrade to an already-existing version, using "`make test`" ensures that you will test your new extension, not the already-existing version.

When Perl sees a `use extension;`, it searches for a file with the same name as the `use`'d extension that has a .pm suffix. If that file cannot be found, Perl dies with a fatal error. The default search path is contained in the `@INC` array.

In our case, Mytest.pm tells perl that it will need the Exporter and Dynamic Loader extensions. It then sets the `@ISA` and `@EXPORT` arrays and the `$VERSION` scalar; finally it tells perl to bootstrap the module. Perl will call its dynamic loader routine (if there is one) and load the shared library.

The two arrays `@ISA` and `@EXPORT` are very important. The `@ISA` array contains a list of other packages in which to search for methods (or subroutines) that do not exist in the current package. This is usually only important for object-oriented extensions (which we will talk about much later), and so usually doesn't need to be modified.

The `@EXPORT` array tells Perl which of the extension's variables and subroutines should be placed into the calling package's namespace. Because you don't know if the user has already used your variable and subroutine names, it's vitally important to carefully select what to export. Do *not* export method or variable names *by default* without a good reason.

As a general rule, if the module is trying to be object-oriented then don't export anything. If it's just a collection of functions and variables, then you can export them via another array, called `@EXPORT_OK`. This array does not automatically place its subroutine and variable names into the namespace unless the user specifically requests that this be done.

See *perlmod* for more information.

The `$VERSION` variable is used to ensure that the .pm file and the shared library are "in sync" with each other. Any time you make changes to the .pm or .xs files, you should increment the value of this variable.

Writing good test scripts

The importance of writing good test scripts cannot be over-emphasized. You should closely follow the "ok/not ok" style that Perl itself uses, so that it is very easy and unambiguous to determine the outcome of each test case. When you find and fix a bug, make sure you add a test case for it.

By running "`make test`", you ensure that your Mytest.t script runs and uses the correct version of your extension. If you have many test cases, save your test files in the "t" directory and use the suffix ".t". When you run "`make test`", all of these test files will be executed.

EXAMPLE 3

Our third extension will take one argument as its input, round off that value, and set the *argument* to the rounded value.

Add the following to the end of Mytest.xs:

```
void
round(arg)
 double  arg
    CODE:
 if (arg > 0.0) {
  arg = floor(arg + 0.5);
 } else if (arg < 0.0) {
  arg = ceil(arg - 0.5);
 } else {
  arg = 0.0;
 }
    OUTPUT:
 arg
```

Edit the Makefile.PL file so that the corresponding line looks like this:

```
'LIBS'      => ['-lm'],   # e.g., '-lm'
```

Generate the Makefile and run make. Change the test number in Mytest.t to "9" and add the following tests:

```
$i = -1.5; &Mytest::round($i); is( $i, -2.0 );
$i = -1.1; &Mytest::round($i); is( $i, -1.0 );
$i = 0.0; &Mytest::round($i);  is( $i,  0.0 );
$i = 0.5; &Mytest::round($i);  is( $i,  1.0 );
$i = 1.2; &Mytest::round($i);  is( $i,  1.0 );
```

Running "`make test`" should now print out that all nine tests are okay.

Notice that in these new test cases, the argument passed to round was a scalar variable. You might be wondering if you can round a constant or literal. To see what happens, temporarily add the following line to Mytest.t:

```
&Mytest::round(3);
```

Run "`make test`" and notice that Perl dies with a fatal error. Perl won't let you change the value of constants!

What's new here?

- We've made some changes to Makefile.PL. In this case, we've specified an extra library to be linked into the extension's shared library, the math library libm in this case. We'll talk later about how to write XSUBs that can call every routine in a library.
- The value of the function is not being passed back as the function's return value, but by changing the value of the variable that was passed into the function. You might have guessed that when you saw that the return value of round is of type "void".

Input and Output Parameters

You specify the parameters that will be passed into the XSUB on the line(s) after you declare the function's return value and name. Each input parameter line starts with optional whitespace, and may have an optional terminating semicolon.

The list of output parameters occurs at the very end of the function, just before after the OUTPUT: directive. The use of RETVAL tells Perl that you wish to send this value back as the return value of the XSUB function. In Example 3, we wanted the "return value" placed in the original variable which we passed in, so we listed it (and not RETVAL) in the OUTPUT: section.

The XSUBPP Program

The **xsubpp** program takes the XS code in the .xs file and translates it into C code, placing it in a file whose suffix is .c. The C code created makes heavy use of the C functions within Perl.

The TYPEMAP file

The **xsubpp** program uses rules to convert from Perl's data types (scalar, array, etc.) to C's data types (int, char, etc.). These rules are stored in the typemap file ($PERLLIB/ExtUtils/typemap). This file is split into three parts.

The first section maps various C data types to a name, which corresponds somewhat with the various Perl types. The second section contains C code which **xsubpp** uses to handle input parameters. The third section contains C code which **xsubpp** uses to handle output parameters.

Let's take a look at a portion of the .c file created for our extension. The file name is Mytest.c:

```
XS(XS_Mytest_round)
{
```

```
    dXSARGS;
    if (items != 1)
  Perl_croak(aTHX_ "Usage: Mytest::round(arg)");
       PERL_UNUSED_VAR(cv); /* -W */
    {
  double  arg = (double)SvNV(ST(0)); /* XXXXX */
  if (arg > 0.0) {
   arg = floor(arg + 0.5);
  } else if (arg < 0.0) {
   arg = ceil(arg - 0.5);
  } else {
   arg = 0.0;
  }
  sv_setnv(ST(0), (double)arg); /* XXXXX */
       SvSETMAGIC(ST(0));
    }
    XSRETURN_EMPTY;
}
```

Notice the two lines commented with "XXXXX". If you check the first section of the typemap file, you'll see that doubles are of type T_DOUBLE. In the INPUT section, an argument that is T_DOUBLE is assigned to the variable arg by calling the routine SvNV on something, then casting it to double, then assigned to the variable arg. Similarly, in the OUTPUT section, once arg has its final value, it is passed to the sv_setnv function to be passed back to the calling subroutine. These two functions are explained in *perlguts*; we'll talk more later about what that "ST(0)" means in the section on the argument stack.

Warning about Output Arguments

In general, it's not a good idea to write extensions that modify their input parameters, as in Example 3. Instead, you should probably return multiple values in an array and let the caller handle them (we'll do this in a later example). However, in order to better accommodate calling pre-existing C routines, which often do modify their input parameters, this behavior is tolerated.

EXAMPLE 4

In this example, we'll now begin to write XSUBs that will interact with pre-defined C libraries. To begin with, we will build a small library of our own, then let h2xs write our .pm and .xs files for us.

Create a new directory called Mytest2 at the same level as the directory Mytest. In the Mytest2 directory, create another directory called mylib, and cd into that directory.

Here we'll create some files that will generate a test library. These will include a C source file and a header file. We'll also create a Makefile.PL in this directory. Then we'll make sure that running make at the Mytest2 level will automatically run this Makefile.PL file and the resulting Makefile.

In the mylib directory, create a file mylib.h that looks like this:

```
#define TESTVAL 4

extern double foo(int, long, const char*);
```

Also create a file mylib.c that looks like this:

```
#include <stdlib.h>
#include "./mylib.h"

double
```

```
foo(int a, long b, const char *c)
{
 return (a + b + atof(c) + TESTVAL);
}
```

And finally create a file Makefile.PL that looks like this:

```
use ExtUtils::MakeMaker;
$Verbose = 1;
WriteMakefile(
    NAME   => 'Mytest2::mylib',
    SKIP   => [qw(all static static_lib dynamic dynamic_lib)],
    clean  => {'FILES' => 'libmylib$(LIB_EXT)'},
);

sub MY::top_targets {
 '
all :: static

pure_all :: static

static ::       libmylib$(LIB_EXT)

libmylib$(LIB_EXT): $(O_FILES)
 $(AR) cr libmylib$(LIB_EXT) $(O_FILES)
 $(RANLIB) libmylib$(LIB_EXT)

';
}
```

Make sure you use a tab and not spaces on the lines beginning with "$(AR)" and "$(RANLIB)". Make will not function properly if you use spaces. It has also been reported that the "cr" argument to $(AR) is unnecessary on Win32 systems.

We will now create the main top-level Mytest2 files. Change to the directory above Mytest2 and run the following command:

```
% h2xs -O -n Mytest2 ./Mytest2/mylib/mylib.h
```

This will print out a warning about overwriting Mytest2, but that's okay. Our files are stored in Mytest2/mylib, and will be untouched.

The normal Makefile.PL that h2xs generates doesn't know about the mylib directory. We need to tell it that there is a subdirectory and that we will be generating a library in it. Let's add the argument MYEXTLIB to the WriteMakefile call so that it looks like this:

```
WriteMakefile(
    'NAME'      => 'Mytest2',
    'VERSION_FROM' => 'Mytest2.pm', # finds $VERSION
    'LIBS'      => [''],   # e.g., '-lm'
    'DEFINE'    => '',     # e.g., '-DHAVE_SOMETHING'
    'INC'       => '',     # e.g., '-I/usr/include/other'
    'MYEXTLIB' => 'mylib/libmylib$(LIB_EXT)',
);
```

and then at the end add a subroutine (which will override the pre-existing subroutine). Remember to use a tab character to indent the line beginning with "cd"!

```
sub MY::postamble {
'
$(MYEXTLIB): mylib/Makefile
 cd mylib && $(MAKE) $(PASSTHRU)
';
}
```

Let's also fix the MANIFEST file so that it accurately reflects the contents of our extension. The single line that says "mylib" should be replaced by the following three lines:

```
mylib/Makefile.PL
mylib/mylib.c
mylib/mylib.h
```

To keep our namespace nice and unpolluted, edit the .pm file and change the variable `@EXPORT` to `@EXPORT_OK`. Finally, in the .xs file, edit the #include line to read:

```
#include "mylib/mylib.h"
```

And also add the following function definition to the end of the .xs file:

```
double
foo(a,b,c)
 int             a
 long            b
 const char *    c
    OUTPUT:
 RETVAL
```

Now we also need to create a typemap file because the default Perl doesn't currently support the const char * type. Create a file called typemap in the Mytest2 directory and place the following in it:

```
const char * T_PV
```

Now run perl on the top-level Makefile.PL. Notice that it also created a Makefile in the mylib directory. Run make and watch that it does cd into the mylib directory and run make in there as well.

Now edit the Mytest2.t script and change the number of tests to "4", and add the following lines to the end of the script:

```
is( &Mytest2::foo(1, 2, "Hello, world!"), 7 );
is( &Mytest2::foo(1, 2, "0.0"), 7 );
ok( abs(&Mytest2::foo(0, 0, "-3.4") - 0.6) <= 0.01 );
```

(When dealing with floating-point comparisons, it is best to not check for equality, but rather that the difference between the expected and actual result is below a certain amount (called epsilon) which is 0.01 in this case)

Run "`make test`" and all should be well. There are some warnings on missing tests for the Mytest2::mylib extension, but you can ignore them.

What has happened here?

Unlike previous examples, we've now run h2xs on a real include file. This has caused some extra goodies to appear in both the .pm and .xs files.

- In the .xs file, there's now a #include directive with the absolute path to the mylib.h header file. We changed this to a relative path so that we could move the extension directory if we wanted to.
- There's now some new C code that's been added to the .xs file. The purpose of the `constant` routine is to make the values that are #define'd in the header file accessible by the Perl script (by calling either `TESTVAL` or `&Mytest2::TESTVAL`). There's also some XS code to allow calls to the `constant` routine.
- The .pm file originally exported the name `TESTVAL` in the `@EXPORT` array. This could lead to name clashes. A good rule of thumb is that if the #define is only going to be used by the C routines themselves, and not by the user, they should be removed from the `@EXPORT` array. Alternately, if you don't mind using the "fully qualified name" of a variable, you could move most or all of the items from the `@EXPORT` array into the `@EXPORT_OK` array.
- If our include file had contained #include directives, these would not have been processed by h2xs. There is no good solution to this right now.
- We've also told Perl about the library that we built in the mylib subdirectory. That required only the addition of the `MYEXTLIB` variable to the WriteMakefile call and the replacement of the postamble subroutine to cd into the subdirectory and run make. The Makefile.PL for the library is a bit more complicated, but not excessively so. Again we replaced the postamble subroutine to insert our own code. This code simply specified that the library to be created here was a static archive library (as opposed to a dynamically loadable library) and provided the commands to build it.

Anatomy of .xs file

The .xs file of *EXAMPLE 4* contained some new elements. To understand the meaning of these elements, pay attention to the line which reads

```
MODULE = Mytest2  PACKAGE = Mytest2
```

Anything before this line is plain C code which describes which headers to include, and defines some convenience functions. No translations are performed on this part, apart from having embedded POD documentation skipped over (see *perlpod*) it goes into the generated output C file as is.

Anything after this line is the description of XSUB functions. These descriptions are translated by **xsubpp** into C code which implements these functions using Perl calling conventions, and which makes these functions visible from Perl interpreter.

Pay a special attention to the function `constant`. This name appears twice in the generated .xs file: once in the first part, as a static C function, then another time in the second part, when an XSUB interface to this static C function is defined.

This is quite typical for .xs files: usually the .xs file provides an interface to an existing C function. Then this C function is defined somewhere (either in an external library, or in the first part of .xs file), and a Perl interface to this function (i.e. "Perl glue") is described in the second part of .xs file. The situation in *EXAMPLE 1*, *EXAMPLE 2*, and *EXAMPLE 3*, when all the work is done inside the "Perl glue", is somewhat of an exception rather than the rule.

Getting the fat out of XSUBs

In *EXAMPLE 4* the second part of .xs file contained the following description of an XSUB:

```
double
foo(a,b,c)
 int             a
 long            b
 const char *    c
   OUTPUT:
```

```
  RETVAL
```

Note that in contrast with *EXAMPLE 1*, *EXAMPLE 2* and *EXAMPLE 3*, this description does not contain the actual *code* for what is done is done during a call to Perl function foo(). To understand what is going on here, one can add a CODE section to this XSUB:

```
double
foo(a,b,c)
 int             a
 long            b
 const char *    c
    CODE:
 RETVAL = foo(a,b,c);
    OUTPUT:
 RETVAL
```

However, these two XSUBs provide almost identical generated C code: **xsubpp** compiler is smart enough to figure out the `CODE:` section from the first two lines of the description of XSUB. What about `OUTPUT:` section? In fact, that is absolutely the same! The `OUTPUT:` section can be removed as well, *as far as `CODE:` section or `PPCODE:` section* is not specified: **xsubpp** can see that it needs to generate a function call section, and will autogenerate the OUTPUT section too. Thus one can shortcut the XSUB to become:

```
double
foo(a,b,c)
 int             a
 long            b
 const char *    c
```

Can we do the same with an XSUB

```
int
is_even(input)
 int input
    CODE:
 RETVAL = (input % 2 == 0);
    OUTPUT:
 RETVAL
```

of *EXAMPLE 2*? To do this, one needs to define a C function `int is_even(int input)`. As we saw in *Anatomy of .xs file*, a proper place for this definition is in the first part of .xs file. In fact a C function

```
int
is_even(int arg)
{
 return (arg % 2 == 0);
}
```

is probably overkill for this. Something as simple as a `#define` will do too:

```
#define is_even(arg) ((arg) % 2 == 0)
```

After having this in the first part of .xs file, the "Perl glue" part becomes as simple as

```
int
```

```
is_even(input)
 int input
```

This technique of separation of the glue part from the workhorse part has obvious tradeoffs: if you want to change a Perl interface, you need to change two places in your code. However, it removes a lot of clutter, and makes the workhorse part independent from idiosyncrasies of Perl calling convention. (In fact, there is nothing Perl-specific in the above description, a different version of **xsubpp** might have translated this to TCL glue or Python glue as well.)

More about XSUB arguments

With the completion of Example 4, we now have an easy way to simulate some real-life libraries whose interfaces may not be the cleanest in the world. We shall now continue with a discussion of the arguments passed to the **xsubpp** compiler.

When you specify arguments to routines in the .xs file, you are really passing three pieces of information for each argument listed. The first piece is the order of that argument relative to the others (first, second, etc). The second is the type of argument, and consists of the type declaration of the argument (e.g., int, char*, etc). The third piece is the calling convention for the argument in the call to the library function.

While Perl passes arguments to functions by reference, C passes arguments by value; to implement a C function which modifies data of one of the "arguments", the actual argument of this C function would be a pointer to the data. Thus two C functions with declarations

```
int string_length(char *s);
int upper_case_char(char *cp);
```

may have completely different semantics: the first one may inspect an array of chars pointed by `s`, and the second one may immediately dereference `cp` and manipulate `*cp` only (using the return value as, say, a success indicator). From Perl one would use these functions in a completely different manner.

One conveys this info to **xsubpp** by replacing `*` before the argument by `&`. `&` means that the argument should be passed to a library function by its address. The above two function may be XSUB-ified as

```
int
string_length(s)
 char * s

int
upper_case_char(cp)
 char &cp
```

For example, consider:

```
int
foo(a,b)
 char &a
 char * b
```

The first Perl argument to this function would be treated as a char and assigned to the variable a, and its address would be passed into the function foo. The second Perl argument would be treated as a string pointer and assigned to the variable b. The *value* of b would be passed into the function foo. The actual call to the function foo that **xsubpp** generates would look like this:

```
foo(&a, b);
```

xsubpp will parse the following function argument lists identically:

```
char &a
char&a
char & a
```

However, to help ease understanding, it is suggested that you place a "&" next to the variable name and away from the variable type), and place a "*" near the variable type, but away from the variable name (as in the call to foo above). By doing so, it is easy to understand exactly what will be passed to the C function -- it will be whatever is in the "last column".

You should take great pains to try to pass the function the type of variable it wants, when possible. It will save you a lot of trouble in the long run.

The Argument Stack

If we look at any of the C code generated by any of the examples except example 1, you will notice a number of references to ST(n), where n is usually 0. "ST" is actually a macro that points to the n'th argument on the argument stack. ST(0) is thus the first argument on the stack and therefore the first argument passed to the XSUB, ST(1) is the second argument, and so on.

When you list the arguments to the XSUB in the .xs file, that tells **xsubpp** which argument corresponds to which of the argument stack (i.e., the first one listed is the first argument, and so on). You invite disaster if you do not list them in the same order as the function expects them.

The actual values on the argument stack are pointers to the values passed in. When an argument is listed as being an OUTPUT value, its corresponding value on the stack (i.e., ST(0) if it was the first argument) is changed. You can verify this by looking at the C code generated for Example 3. The code for the round() XSUB routine contains lines that look like this:

```
double  arg = (double)SvNV(ST(0));
/* Round the contents of the variable arg */
sv_setnv(ST(0), (double)arg);
```

The arg variable is initially set by taking the value from ST(0), then is stored back into ST(0) at the end of the routine.

XSUBs are also allowed to return lists, not just scalars. This must be done by manipulating stack values ST(0), ST(1), etc, in a subtly different way. See *perlxs* for details.

XSUBs are also allowed to avoid automatic conversion of Perl function arguments to C function arguments. See *perlxs* for details. Some people prefer manual conversion by inspecting `ST(i)` even in the cases when automatic conversion will do, arguing that this makes the logic of an XSUB call clearer. Compare with *Getting the fat out of XSUBs* for a similar tradeoff of a complete separation of "Perl glue" and "workhorse" parts of an XSUB.

While experts may argue about these idioms, a novice to Perl guts may prefer a way which is as little Perl-guts-specific as possible, meaning automatic conversion and automatic call generation, as in *Getting the fat out of XSUBs*. This approach has the additional benefit of protecting the XSUB writer from future changes to the Perl API.

Extending your Extension

Sometimes you might want to provide some extra methods or subroutines to assist in making the interface between Perl and your extension simpler or easier to understand. These routines should live in the .pm file. Whether they are automatically loaded when the extension itself is loaded or only loaded when called depends on where in the .pm file the subroutine definition is placed. You can also consult *AutoLoader* for an alternate way to store and load your extra subroutines.

Documenting your Extension

There is absolutely no excuse for not documenting your extension. Documentation belongs in the .pm file. This file will be fed to pod2man, and the embedded documentation will be converted to the manpage format, then placed in the blib directory. It will be copied to Perl's manpage directory when the extension is installed.

You may intersperse documentation and Perl code within the .pm file. In fact, if you want to use method autoloading, you must do this, as the comment inside the .pm file explains.

See *perlpod* for more information about the pod format.

Installing your Extension

Once your extension is complete and passes all its tests, installing it is quite simple: you simply run "make install". You will either need to have write permission into the directories where Perl is installed, or ask your system administrator to run the make for you.

Alternately, you can specify the exact directory to place the extension's files by placing a "PREFIX=/destination/directory" after the make install. (or in between the make and install if you have a brain-dead version of make). This can be very useful if you are building an extension that will eventually be distributed to multiple systems. You can then just archive the files in the destination directory and distribute them to your destination systems.

EXAMPLE 5

In this example, we'll do some more work with the argument stack. The previous examples have all returned only a single value. We'll now create an extension that returns an array.

This extension is very Unix-oriented (struct statfs and the statfs system call). If you are not running on a Unix system, you can substitute for statfs any other function that returns multiple values, you can hard-code values to be returned to the caller (although this will be a bit harder to test the error case), or you can simply not do this example. If you change the XSUB, be sure to fix the test cases to match the changes.

Return to the Mytest directory and add the following code to the end of Mytest.xs:

```
void
statfs(path)
 char *  path
    INIT:
 int i;
 struct statfs buf;

    PPCODE:
 i = statfs(path, &buf);
 if (i == 0) {
  XPUSHs(sv_2mortal(newSVnv(buf.f_bavail)));
  XPUSHs(sv_2mortal(newSVnv(buf.f_bfree)));
  XPUSHs(sv_2mortal(newSVnv(buf.f_blocks)));
  XPUSHs(sv_2mortal(newSVnv(buf.f_bsize)));
  XPUSHs(sv_2mortal(newSVnv(buf.f_ffree)));
  XPUSHs(sv_2mortal(newSVnv(buf.f_files)));
  XPUSHs(sv_2mortal(newSVnv(buf.f_type)));
 } else {
  XPUSHs(sv_2mortal(newSVnv(errno)));
 }
```

You'll also need to add the following code to the top of the .xs file, just after the include of "XSUB.h":

```
#include <sys/vfs.h>
```

Also add the following code segment to Mytest.t while incrementing the "9" tests to "11":

```
@a = &Mytest::statfs("/blech");
ok( scalar(@a) == 1 && $a[0] == 2 );
@a = &Mytest::statfs("/");
is( scalar(@a), 7 );
```

New Things in this Example

This example added quite a few new concepts. We'll take them one at a time.

- The INIT: directive contains code that will be placed immediately after the argument stack is decoded. C does not allow variable declarations at arbitrary locations inside a function, so this is usually the best way to declare local variables needed by the XSUB. (Alternatively, one could put the whole `PPCODE:` section into braces, and put these declarations on top.)

- This routine also returns a different number of arguments depending on the success or failure of the call to statfs. If there is an error, the error number is returned as a single-element array. If the call is successful, then a 9-element array is returned. Since only one argument is passed into this function, we need room on the stack to hold the 9 values which may be returned.

  We do this by using the PPCODE: directive, rather than the CODE: directive. This tells **xsubpp** that we will be managing the return values that will be put on the argument stack by ourselves.

- When we want to place values to be returned to the caller onto the stack, we use the series of macros that begin with "XPUSH". There are five different versions, for placing integers, unsigned integers, doubles, strings, and Perl scalars on the stack. In our example, we placed a Perl scalar onto the stack. (In fact this is the only macro which can be used to return multiple values.)

  The XPUSH* macros will automatically extend the return stack to prevent it from being overrun. You push values onto the stack in the order you want them seen by the calling program.

- The values pushed onto the return stack of the XSUB are actually mortal SV's. They are made mortal so that once the values are copied by the calling program, the SV's that held the returned values can be deallocated. If they were not mortal, then they would continue to exist after the XSUB routine returned, but would not be accessible. This is a memory leak.

- If we were interested in performance, not in code compactness, in the success branch we would not use `XPUSHs` macros, but `PUSHs` macros, and would pre-extend the stack before pushing the return values:

  ```
  EXTEND(SP, 7);
  ```

  The tradeoff is that one needs to calculate the number of return values in advance (though overextending the stack will not typically hurt anything but memory consumption).

  Similarly, in the failure branch we could use `PUSHs` *without* extending the stack: the Perl function reference comes to an XSUB on the stack, thus the stack is *always* large enough to take one return value.

EXAMPLE 6

In this example, we will accept a reference to an array as an input parameter, and return a reference to an array of hashes. This will demonstrate manipulation of complex Perl data types from an XSUB.

This extension is somewhat contrived. It is based on the code in the previous example. It calls the statfs function multiple times, accepting a reference to an array of filenames as input, and returning a

reference to an array of hashes containing the data for each of the filesystems.

Return to the Mytest directory and add the following code to the end of Mytest.xs:

```
    SV *
    multi_statfs(paths)
     SV * paths
 INIT:
     AV * results;
     I32 numpaths = 0;
     int i, n;
     struct statfs buf;

     if ((!SvROK(paths))
  || (SvTYPE(SvRV(paths)) != SVt_PVAV)
  || ((numpaths = av_len((AV *)SvRV(paths))) < 0))
     {
  XSRETURN_UNDEF;
     }
     results = (AV *)sv_2mortal((SV *)newAV());
 CODE:
     for (n = 0; n <= numpaths; n++) {
  HV * rh;
  STRLEN l;
  char * fn = SvPV(*av_fetch((AV *)SvRV(paths), n, 0), l);

  i = statfs(fn, &buf);
  if (i != 0) {
      av_push(results, newSVnv(errno));
      continue;
  }

  rh = (HV *)sv_2mortal((SV *)newHV());

  hv_store(rh, "f_bavail", 8, newSVnv(buf.f_bavail), 0);
  hv_store(rh, "f_bfree",  7, newSVnv(buf.f_bfree),  0);
  hv_store(rh, "f_blocks", 8, newSVnv(buf.f_blocks), 0);
  hv_store(rh, "f_bsize",  7, newSVnv(buf.f_bsize),  0);
  hv_store(rh, "f_ffree",  7, newSVnv(buf.f_ffree),  0);
  hv_store(rh, "f_files",  7, newSVnv(buf.f_files),  0);
  hv_store(rh, "f_type",   6, newSVnv(buf.f_type),   0);

  av_push(results, newRV((SV *)rh));
     }
     RETVAL = newRV((SV *)results);
 OUTPUT:
     RETVAL
```

And add the following code to Mytest.t, while incrementing the "11" tests to "13":

```
$results = Mytest::multi_statfs([ '/', '/blech' ]);
ok( ref $results->[0]) );
ok( ! ref $results->[1] );
```

New Things in this Example

There are a number of new concepts introduced here, described below:

- This function does not use a typemap. Instead, we declare it as accepting one SV* (scalar) parameter, and returning an SV* value, and we take care of populating these scalars within the code. Because we are only returning one value, we don't need a `PPCODE:` directive - instead, we use `CODE:` and `OUTPUT:` directives.

- When dealing with references, it is important to handle them with caution. The `INIT:` block first checks that `SvROK` returns true, which indicates that paths is a valid reference. It then verifies that the object referenced by paths is an array, using `SvRV` to dereference paths, and `SvTYPE` to discover its type. As an added test, it checks that the array referenced by paths is non-empty, using the `av_len` function (which returns -1 if the array is empty). The XSRETURN_UNDEF macro is used to abort the XSUB and return the undefined value whenever all three of these conditions are not met.

- We manipulate several arrays in this XSUB. Note that an array is represented internally by an AV* pointer. The functions and macros for manipulating arrays are similar to the functions in Perl: `av_len` returns the highest index in an AV*, much like $#array; `av_fetch` fetches a single scalar value from an array, given its index; `av_push` pushes a scalar value onto the end of the array, automatically extending the array as necessary.

  Specifically, we read pathnames one at a time from the input array, and store the results in an output array (results) in the same order. If statfs fails, the element pushed onto the return array is the value of errno after the failure. If statfs succeeds, though, the value pushed onto the return array is a reference to a hash containing some of the information in the statfs structure.

  As with the return stack, it would be possible (and a small performance win) to pre-extend the return array before pushing data into it, since we know how many elements we will return:

```
av_extend(results, numpaths);
```

- We are performing only one hash operation in this function, which is storing a new scalar under a key using `hv_store`. A hash is represented by an HV* pointer. Like arrays, the functions for manipulating hashes from an XSUB mirror the functionality available from Perl. See *perlguts* and *perlapi* for details.

- To create a reference, we use the `newRV` function. Note that you can cast an AV* or an HV* to type SV* in this case (and many others). This allows you to take references to arrays, hashes and scalars with the same function. Conversely, the `SvRV` function always returns an SV*, which may need to be cast to the appropriate type if it is something other than a scalar (check with `SvTYPE`).

- At this point, xsubpp is doing very little work - the differences between Mytest.xs and Mytest.c are minimal.

EXAMPLE 7 (Coming Soon)

XPUSH args AND set RETVAL AND assign return value to array

EXAMPLE 8 (Coming Soon)

Setting $!

EXAMPLE 9 Passing open files to XSes

You would think passing files to an XS is difficult, with all the typeglobs and stuff. Well, it isn't.

Suppose that for some strange reason we need a wrapper around the standard C library function `fputs()`. This is all we need:

```
#define PERLIO_NOT_STDIO 0
```

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include <stdio.h>

int
fputs(s, stream)
 char *          s
 FILE *         stream
```

The real work is done in the standard typemap.

But you loose all the fine stuff done by the perlio layers. This calls the stdio function `fputs()`, which knows nothing about them.

The standard typemap offers three variants of PerlIO *: `InputStream` (T_IN), `InOutStream` (T_INOUT) and `OutputStream` (T_OUT). A bare `PerlIO *` is considered a T_INOUT. If it matters in your code (see below for why it might) #define or typedef one of the specific names and use that as the argument or result type in your XS file.

The standard typemap does not contain PerlIO * before perl 5.7, but it has the three stream variants. Using a PerlIO * directly is not backwards compatible unless you provide your own typemap.

For streams coming *from* perl the main difference is that `OutputStream` will get the output PerlIO * - which may make a difference on a socket. Like in our example...

For streams being handed *to* perl a new file handle is created (i.e. a reference to a new glob) and associated with the PerlIO * provided. If the read/write state of the PerlIO * is not correct then you may get errors or warnings from when the file handle is used. So if you opened the PerlIO * as "w" it should really be an `OutputStream` if open as "r" it should be an `InputStream`.

Now, suppose you want to use perlio layers in your XS. We'll use the perlio `PerlIO_puts()` function as an example.

In the C part of the XS file (above the first MODULE line) you have

```
#define OutputStream PerlIO *
   or
typedef PerlIO * OutputStream;
```

And this is the XS code:

```
int
perlioputs(s, stream)
 char *          s
 OutputStream stream
CODE:
 RETVAL = PerlIO_puts(stream, s);
OUTPUT:
 RETVAL
```

We have to use a `CODE` section because `PerlIO_puts()` has the arguments reversed compared to `fputs()`, and we want to keep the arguments the same.

Wanting to explore this thoroughly, we want to use the stdio `fputs()` on a PerlIO *. This means we have to ask the perlio system for a stdio `FILE *`:

```
int
perliofputs(s, stream)
 char *          s
 OutputStream stream
PREINIT:
 FILE *fp = PerlIO_findFILE(stream);
CODE:
 if (fp != (FILE*) 0) {
  RETVAL = fputs(s, fp);
 } else {
  RETVAL = -1;
 }
OUTPUT:
 RETVAL
```

Note: `PerlIO_findFILE()` will search the layers for a stdio layer. If it can't find one, it will call `PerlIO_exportFILE()` to generate a new stdio `FILE`. Please only call `PerlIO_exportFILE()` if you want a *new* `FILE`. It will generate one on each call and push a new stdio layer. So don't call it repeatedly on the same file. `PerlIO()`_findFILE will retrieve the stdio layer once it has been generated by `PerlIO_exportFILE()`.

This applies to the perlio system only. For versions before 5.7, `PerlIO_exportFILE()` is equivalent to `PerlIO_findFILE()`.

Troubleshooting these Examples

As mentioned at the top of this document, if you are having problems with these example extensions, you might see if any of these help you.

- In versions of 5.002 prior to the gamma version, the test script in Example 1 will not function properly. You need to change the "use lib" line to read:

  ```
  use lib './blib';
  ```

- In versions of 5.002 prior to version 5.002b1h, the test.pl file was not automatically created by h2xs. This means that you cannot say "make test" to run the test script. You will need to add the following line before the "use extension" statement:

  ```
  use lib './blib';
  ```

- In versions 5.000 and 5.001, instead of using the above line, you will need to use the following line:

  ```
  BEGIN { unshift(@INC, "./blib") }
  ```

- This document assumes that the executable named "perl" is Perl version 5. Some systems may have installed Perl version 5 as "perl5".

See also

For more information, consult *perlguts*, *perlapi*, *perlxs*, *perlmod*, and *perlpod*.

Author

Jeff Okamoto <*okamoto@corp.hp.com*>

Reviewed and assisted by Dean Roehrich, Ilya Zakharevich, Andreas Koenig, and Tim Bunce.

PerlIO material contributed by Lupe Christoph, with some clarification by Nick Ing-Simmons.

Changes for h2xs as of Perl 5.8.x by Renee Baecker

Last Changed

2007/10/11